Interprétation et Compilation (HLIN604)

Michel Meynard

14 janvier 2016

Table des matières

Avant-propos **v**

1 Introduction **1**
1.1 Objectifs . . . 1
1.2 Bibliographie . . . 1
1.3 Rappels théoriques . . . 1
1.3.1 Familles de grammaires et de langages : hiérarchie de Chomsky . . . 1
1.3.2 Langages réguliers : propriétés et caractérisations . . . 2
1.3.3 Langages algébriques : propriétés et caractérisations . . . 2
1.4 Types de traducteurs . . . 2
1.5 Modèle classique de compilation . . . 3
1.6 Remarques . . . 3
1.7 Vocabulaire des langages de programmation . . . 3

2 Analyse lexicale **5**
2.1 Reconnaissance d'un mot par un AFD . . . 5
2.2 Implémentation des Automates Finis Déterministes AFD . . . 5
2.3 Analyseur lexical . . . 7
2.4 Implémentation des analyseurs lexicaux . . . 9
2.5 Un langage et un outil pour l'analyse lexicale : (f)lex . . . 10
2.5.1 Un exemple . . . 10
2.5.2 Syntaxe et sémantique des sources Flex . . . 11
2.5.3 La commande flex . . . 14
2.5.4 Actions C++ . . . 15
2.5.5 Liaison avec un analyseur syntaxique . . . 15
2.6 Algorithmique . . . 15
2.6.1 Traduction des expressions régulières . . . 15
2.6.2 Déterminisation . . . 16
2.6.3 Minimisation . . . 17

3 Analyse syntaxique **19**
3.1 Analyse descendante récursive . . . 19
3.2 Analyse descendante par automate à pile . . . 21
3.2.1 Introduction . . . 21
3.2.2 Fonctionnement de l'automate à pile en analyse descendante . . . 21
3.2.3 Algorithmique . . . 22
3.2.4 Construction de la table d'analyse . . . 29
3.2.5 Grammaires LL(1) . . . 30
3.2.6 Conclusion sur l'analyse descendante . . . 30
3.3 Un langage et un outil pour l'analyse syntaxique : yacc . . . 31
3.3.1 Un exemple . . . 31
3.3.2 Syntaxe et sémantique des sources yacc . . . 32
3.3.3 Un exemple complet : une calculette . . . 36
3.3.4 Bison (version 2.3) et analyseur C++ . . . 38
3.4 Analyse ascendante par automate à pile . . . 41
3.4.1 Fonctionnement de l'automate à pile en analyse ascendante LR . . . 42
3.4.2 Algorithmique . . . 43
3.5 Les conflits et leur résolution par yacc . . . 46

4 Analyse sémantique **49**
4.1 Table des symboles . . . 49
4.1.1 Généralités . . . 49
4.1.2 Implémentation d'une table des symboles . . . 49
4.2 Gestion des erreurs . . . 50
4.3 Arbre abstrait . . . 50
4.4 Traduction dirigée par la syntaxe . . . 50
4.4.1 Grammaires attribuées . . . 50
4.4.2 Méthode de transformation des grammaires L-attribuées . . . 54

5 Génération de code **57**
5.1 Introduction . . . 57
5.2 Machine virtuelle à pile . . . 57
5.3 Développement d'un compilateur . . . 57

6 Sémantique opérationnelle des langages de programmation **59**
6.1 Introduction . . . 59
6.2 Organisation de l'espace mémoire . . . 59
6.2.1 Image mémoire . . . 59
6.2.2 Appel procédural . . . 59
6.2.3 Passage des paramètres . . . 60
6.2.4 Accès aux noms (liaison) . . . 60
6.3 Langages à objets . . . 60
6.3.1 C++ . . . 60
6.3.2 Java . . . 62

Index **63**

Bibliographie **64**

Solutions des exercices **67**

Avant-propos

Ce cours s'inscrit dans un enseignement d'informatique, discipline scientifique dispensée à la faculté des sciences de l'université de Montpellier. L'évolution des connaissances scientifiques dans l'histoire depuis la Grèce antique jusqu'à nos jours a souvent remis en cause des connaissances plus anciennes ou des dogmes religieux. Au IVe siècle avant notre ère, le grec Anaxagore de Clazomènes est le premier savant de l'histoire à être accusé d'impiété par les autorités religieuses de l'époque ["Bruno et Galilée au regard de l'infini" de Jean-Pierre Luminet]. Il ne doit sa liberté qu'à son amitié avec Périclès. Plus tard, Giordano Bruno (1548-1600) invente la cosmologie infinitiste mais également le principe d'inertie. Ses idées et ses écrits contraires à la doctrine chrétienne le conduisent à être incarcéré au total pendant huit ans dans les geôles de l'Inquisition. Après de multiples procès, le 16 Février de l'an de grâce 1600, Giordano BRUNO est torturé et brûlé vif, par l'inquisition catholique, à Rome, sur le Campo dei Fiori, pour avoir refusé d'abjurer ses idées. Plus tard, Galilée, Kepler auront également des problèmes juridiques liés à l'expression de leurs idées scientifiques révolutionnaires.

En France, le siècle des lumières puis la révolution française de 1789 ont permis de donner la liberté d'expression aux scientifiques (et aux autres) afin que leurs travaux de recherche puissent être publiés, discutés, réfutés ou approuvés. La loi de séparation des Églises et de l'État a été adoptée le 9 décembre 1905 à l'initiative du député républicain-socialiste Aristide Briand. Elle prend parti en faveur d'une laïcité sans excès. Elle est avant tout un acte fondateur dans l'affrontement violent qui a opposé deux conceptions sur la place des Églises dans la société française pendant presque vingt-cinq ans.

La liberté de pensée et d'expression constituent donc les fondements d'un enseignement universitaire de qualité. D'autres part, les scientifiques étant des citoyens comme les autres, il convient de rappeler quelques lois françaises qui nous gouvernent.

Quelques lois fondamentales

Art. 1 de la constitution du 4 octobre 1958 La France est une République indivisible, laïque, démocratique et sociale. Elle assure l'égalité devant la loi de tous les citoyens sans distinction d'origine, de race ou de religion. Elle respecte toutes les croyances. Son organisation est décentralisée. La loi favorise l'égal accès des femmes et des hommes aux mandats électoraux et fonctions électives, ainsi qu'aux responsabilités professionnelles et sociales.

Art. 4 de la Déclaration des Droits de l'Homme et du Citoyen de 1789 La liberté consiste à pouvoir faire tout ce qui ne nuit pas à autrui : ainsi, l'exercice des droits naturels de chaque homme n'a de bornes que celles qui assurent aux autres Membres de la Société la jouissance de ces mêmes droits. Ces bornes ne peuvent être déterminées que par la Loi.

Art. 5 de la Déclaration des Droits de l'Homme et du Citoyen de 1789 La Loi n'a le droit de défendre que les actions nuisibles à la Société. Tout ce qui n'est pas défendu par la Loi ne peut être empêché, et nul ne peut être contraint à faire ce qu'elle n'ordonne pas.

Art. 10 de la Déclaration des Droits de l'Homme et du Citoyen de 1789 Nul ne doit être inquiété pour ses opinions, même religieuses, pourvu que leur manifestation ne trouble pas l'ordre public établi par la Loi.

Art. 11 de la Déclaration des Droits de l'Homme et du Citoyen de 1789 La libre communication des pensées et des opinions est un des droits les plus précieux de l'Homme : tout Citoyen peut donc parler, écrire, imprimer librement, sauf à répondre de l'abus de cette liberté dans les cas déterminés par la Loi.

Chapitre 1

Introduction

1.1 Objectifs

— Mise en oeuvre de la théorie des langages formels.
— Compréhension des techniques de compilation.
— Utilisation d'outils de génération de code (flex, bison).
— Utilité des traducteurs : compilateurs, interpréteurs, convertisseurs de format (rtfToLatex, LaTeXToHtml, postscript To ...).
— Réalisation d'un projet : compilateur d'un langage à objets "Sool".

1.2 Bibliographie

Vous trouverez en fin d'ouvrage, un certain nombre de références de livres vous permettant d'aller plus loin dans l'étude des compilateurs et des langages de programmation. La bible de ces ouvrages [1] est extrêmement complet et détaille les techniques algorithmiques à appliquer aux langages. Tout au contraire, l'ouvrage [2] est une étude mathématique des langages formels. En ce qui concerne le langage Java, le livre de référence [3] décrit le langage et sa bibliothèque fournie tandis que [4] décrit le byte-code. Quant à C++, les livres [5, 6] décrivent le modèle objet de ce langage. Enfin, ne pas oublier le livre [7] qui est un manuel technique de ces deux outils.

1.3 Rappels théoriques

1.3.1 Familles de grammaires et de langages : hiérarchie de Chomsky

On classe les grammaires $G = (V_T, V_N, R, S)$ en quatres grandes familles (ou types ou classes) numérotés de 0 à 3, de la plus large à la plus petite au sens de l'inclusion stricte. Les quatres familles se distinguent par les restrictions imposées aux règles de production de chaque famille.

Type 0 aucune restriction. Les langages engendrés sont qualifiés de **récursivement énumérables**.

Type 1 toute règle r de R est de la forme : $r = \alpha X \beta \to \alpha m \beta$ avec $\alpha, \beta \in V^*$; $X \in V_N$; $m \in V^+$.
Attention m ne peut être le mot vide ! Ces grammaires sont dites **contextuelles** ou dépendant du contexte (α et β représentant ce contexte). Le mot vide ne pouvant être généré par ces grammaires, une exception existe : la règle $S \to \varepsilon$ peut exister à condition que S ne soit pas présente dans une partie droite d'une règle de production.

Exemple 1
le P garçon → le petit garçon ; la P N → la petite N ; N → fille.

Type 2 toute règle r de R est de la forme : $r = X \to \alpha$ avec $\alpha \in V^*$; $X \in V_N$.
Ces grammaires sont dites **algébriques**, ou indépendantes du contexte ("context-free"), ou grammaires de Chomsky, ou C-grammaires.

Exemple 2
$P \to (P)|\varepsilon|PP$ *: une grammaire de parenthèses.*

Type 3 toute règle r de R est de la forme : $r = X \to \alpha$ avec $\alpha \in V_T V_N \cup V_T \cup \{\varepsilon\}$; $X \in V_N$;
Ces grammaires sont dites **régulières**, ou rationnelles, ou grammaires de Kleene, ou K-grammaires.

Exemple 3
$P \to 0|1E|2E| \ldots |9E$ *;* $E \to 0E| \ldots |9E|\varepsilon$ *: une grammaire régulière d'indices.*

Théorème 1 *On note $\mathcal{L}_i$ l'ensemble des langages engendrés par les grammaires de type i. On a alors l'inclusion stricte : $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$.*

1.3.2 Langages réguliers : propriétés et caractérisations

Théorème 2 *Les 4 propositions suivantes sont équivalentes :*

1. *le langage L est défini par une expression régulière ;*
2. *le langage L est généré par une grammaire régulière ;*
3. *le langage L est reconnu par un automate fini déterministe ;*
4. *le langage L est reconnu par un automate fini indéterministe.*

Théorème 3 (Théorème de Kleene) *La famille des langages réguliers $\mathcal{L}_3$ est la plus petite famille de langages qui contient les langages finis et qui est fermée pour les opérations réunion, produit et étoile.*

Théorème 4 (la pompe, version 2a) *Soit L, un langage régulier infini sur V. Alors, $\exists k \in \mathbb{N} - \{0\}$ tel que $\forall m \in L, |m| \geq k, \exists x, u, y \in V^*$ tel que $u \neq \varepsilon, m = xuy, |xu| \leq k$ et $\forall n \in \mathbb{N}, xu^n y \in L$.*

Théorème 5 *Le langage inverse, complémentaire d'un langage régulier est régulier. L'intersection de deux langages réguliers est régulier.*

1.3.3 Langages algébriques : propriétés et caractérisations

Définition 1 *L'ensemble des arbres de dérivation (ou arbres syntaxiques) associé à une grammaire $G = (V_T, V_N, R, S)$, noté $\mathcal{A}(G)$ est un esemble d'arbres étiquetés construits par le schéma d'induction suivant.*

Univers *Ensemble de tous les arbres dont les nœuds sont étiquetés par des symbole de $V \cup \{\varepsilon\}$.*
Base *Ensemble de tous les arbres réduits à une unique racine étiquetée par un symbole de $V \cup \{\varepsilon\}$.*
Règles *Soit une règle de production quelconque $X \to y_1 y_2 \dots y_n$ avec $X \in V_N, y_i \in V \cup \{\varepsilon\}$. Soient n arbres syntaxiques $a_1, a_2, \dots, a_n$ dont les racines sont étiquetées par $y_1, y_2, \dots, y_n$. Alors l'arbre de racine étiquetée par X et de sous-arbres $a_1, a_2, \dots, a_n$ est un arbre de dérivation de G.*

Théorème 6 *L'ensemble des dérivations gauches d'une grammaire algébrique $G = (V_T, V_N, R, S)$ est équipotent à $\mathcal{A}(G)$.*

Définition 2 *Une grammaire $G = (V_T, V_N, R, S)$ est ambiguë si et seulement s'il existe deux dérivations gauches distinctes partant de S et aboutissant au même mot terminal m.*

Théorème 7 *Tout langage régulier est non ambigu.*

Théorème 8 (d'Ogden) *Soit L un langage algébrique infini sur V. Alors, $\exists k \in \mathbb{N} - \{0\}$ tel que $\forall m \in L, |m| > k, \exists x, u, y, v, z \in V^*$ tel que $uv \neq \varepsilon, m = xuyvz, |uyv| \leq k$ et $\forall n \in \mathbb{N}, xu^n yv^n z \in L$.*

Théorème 9 *La famille des langages algébriques $\mathcal{L}_2$ est fermée pour l'union, la concaténation, l'opération *.*

Théorème 10 *La famille des langages algébriques $\mathcal{L}_2$ n'est pas fermée pour l'intersection ni la complémentation.*

1.4 Types de traducteurs

— Préprocesseurs (macro, directives).
— Assembleurs (pentium x86, DEC alpha, ...).
— Compilateurs (C, C++, javac, visual Basic, ...).
— Interpréteurs (basic, shells Unix, SQL, java, ...).
— Convertisseurs (dvips, asciiToPostscript, rtfToLaTeX, ...).

1.5 Modèle classique de compilation

1. Analyse du source :
   (a) lexicale : découpage en "jetons" (tokens) ;
   (b) syntaxique : vérification de la correction grammaticale et production d'une représentation intermédiaire (souvent un arbre) ;
   (c) sémantique : vérification de la correction sémantique du programme (contrôle de type (conversions), non déclarations, protection de composants (privé, public), ...).

   L'analyse génère une table des symboles qui sera utilisée tout au long du processus de compilation. De plus, l'apparition d'erreurs dans chaque phase peut interrompre le processus ou générer des messages d'avertissements ("warnings").
2. Synthèse de la cible :
   (a) génération de code intermédiaire : machine abstraite (ou virtuelle), p-code du Pascal, byte-code de java, basic tokenisé de Visual Basic, ... ;
   (b) optimisation de code : optimiseur de requêtes SQL, optimiseurs C et C++, ... ;
   (c) génération de code cible : langage machine (C, C++), ou autre.

   A la fin de ce processus, il reste encore :
   — soit à lier les différents fichiers objets et bibliothèques (C, C++) en un fichier exécutable (code machine translatable). Le chargeur du système d'exploitation n'aura plus qu'à créer un processus en mémoire centrale, lui allouer les ressources mémoires nécessaires, puis lancer son exécution. Attention, certaines liaisons (linking) peuvent être retardées jusqu'à l'exécution (DLL Microsoft, ELF Unix).
   — Soit à interpréter le code cible. C'est la solution choisie par le langage Java. Cela permet au compilateur javac de générer un code cible indépendant de la plateforme. Il suffit qu'un interprète java (dépendant de la plateforme) soit installé pour exécuter un fichier cible (un .class). Les navigateurs ("browser" Netscape ou Internet Explorer) contiennent tous un interprète intégré ce qui leur permet d'exécuter les "applets" java.

1.6 Remarques

— L'analyse lexicale est souvent réalisée "à la demande" de l'analyse syntaxique, jeton par jeton. Ainsi la décomposition en phase (analyse lexicale, syntaxique, sémantique, ...) n'engendre pas forcément la même décomposition en "passes", une passe correspondant à la lecture séquentielle du résultat de la phase précédente. Les problèmes de "référence en avant" ("forward reference") pose tout de même des problèmes à la compilation en une seule passe. Il faut pouvoir laisser des "blancs" qu'on pourra reprendre plus tard quand on connaîtra la valeur de cette référence.
— Le compilateur est souvent décomposé en une partie "frontale" indépendante de la plateforme de développement, et une partie "finale" dépendante de la plateforme de développement. Ainsi, l'écriture d'un compilateur du même langage source pour une autre plateforme est moins couteuse.

1.7 Vocabulaire des langages de programmation

On définit ci-après un certain nombre de concepts linguistiques fondamentaux dans l'étude des langages de programmation. Bien entendu, selon le langage (C, Python, langage d'assemblage, ...), les différences sont énormes aussi nous nous référerons principalement aux langages "à la C" :

mot-clé (keyword) mot réservé par un langage et qui ne peut être utilisé pour identifier une variable, une fonction, ... Exemples : `if, while, do, class`

identificateur nom d'un objet de programmation (variable, fonction, classe, méthode,...) généralement composé d'une lettre suivie de chiffres et/ou de lettres. Exemples : `i, Etudiant, _Etudiant_h, factorielle`

littéral valeur possible d'un type exprimée littéralement dans le code. Exemples de littéraux :

entiers `123, 0xFF, 0`

chaînes de caractère `"Bonjour", "", "Monsieur %s,\n"`

flottants `13.5, .12, 0.`

booléens `true, false`

instruction (statement) constituant de base d'un programme qui est généralement une instruction. On distingue les constructions : alternative (`if then else`), itératives (`for, while, do`), les expressions, l'instruction vide (`;`).

expression construction syntaxique ayant une valeur. Exemples : `3*i, j++, char**, fact(12)`. Selon les langages, l'affectation est une expression (on peut alors réaliser des affectations multiples `i=j=5`) ou bien une instruction `let i = 5`.

opérateur il permet de construire une expression complexe à partir d'expressions de base (littéraux, identificateurs). Exemples : `+`, `-`, `/`, `*`, `++`, `[]`, `()`

bloc suite d'instructions généralement entouré d'accolades `{...}`.

définition de type, de variable, de fonction, de classe : elle permet de spécifier un objet de programmation.

Chapitre 2

Analyse lexicale

Avant d'aborder l'analyse lexicale, rappelons les résultats sur les Automates d'états Finis Déterministes (AFD).

2.1 Reconnaissance d'un mot par un AFD

Rappelons qu'un AFD possède un unique état initial et aucun couple de transitions $(e_i, a, e_j), (e_i, a, e_k)$ tels que $j \neq k$. Ainsi, l'ensemble des transitions peut être implémenté simplement par une table à double entrée : $TRANS[etatCourant][carCourant]$. **L'algorithme 1** en page 5 décrit la reconnaissance d'un mot par un AFD.

Algorithme 1 : Reconnaissance d'un mot par un AFD

Données : $B = (V, E, D = \{d\}, A, T)$ un AFD ; *mot* une chaîne de caractères ou un flot
Résultat : Booléen
Fonction `accepter(`*B, mot*`)` : Booléen;
début
  etat=d;
  tant que *(c=carSuivant(mot))*$\neq$ \$ **faire**
    si $\exists e \in E$ *tel que* $(etat, c, e) \in T$ **alors**
      etat=e;
    sinon
      retourner *FAUX*;
  retourner $test(etat \in A)$;

2.2 Implémentation des Automates Finis Déterministes AFD

L'implémentation la plus simple d'un AFD consiste à construire la table de transitions dans un tableau. Le programme C de l'exemple 4 illustre un AFD reconnaissant l'expression régulière $a(b^+c)?|bd$.

Exemple 4
Soit l'AFD suivant :

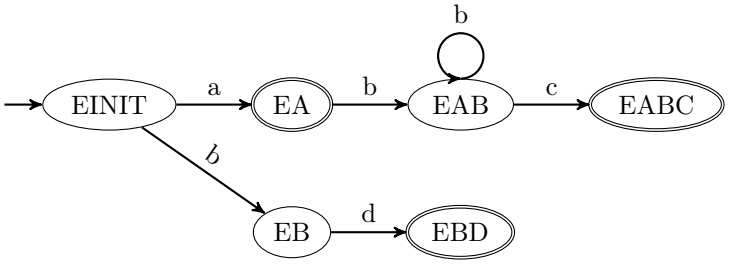

Figure 2.1 – AFD

Nous le représentons par le fichier d'en-tête C suivant :

```
/**@file afd.h
 *@author Michel Meynard
 *@brief Définition d'un AFD
 */
#define EINIT 0
#define EA 1
#define EAB 2
#define EABC 3
#define EB 4
#define EBD 5
#define NBETAT 6

int TRANS[NBETAT][256]; /* table de transition : état suivant */
int FINAL[NBETAT]; /* final (1) ou non (0) ? */

void creerAfd(){ /* Construction de l'AFD */
  int i;int j; /* variables locales */
  for (i=0;i<NBETAT;i++){
    for(j=0;j<256;j++) TRANS[i][j]=-1; /* init vide */
    FINAL[i]=0; /* init tous états non finaux */
  }
  /* Transitions de l'AFD */
  TRANS[EINIT]['a']=EA;TRANS[EA]['b']=EAB;TRANS[EAB]['b']=EAB;
  TRANS[EAB]['c']=EABC;TRANS[EINIT]['b']=EB;TRANS[EB]['d']=EBD;
  FINAL[EA]=FINAL[EABC]=FINAL[EBD]=1; /* états finaux */
}
```

L'implémentation de l'algorithme 1 de reconnaissance est codé dans le fichier C suivant.

```
/**@file accepter.c
 *@author Michel Meynard
 *@brief Définition de la fon accepter
 */
#include <stdio.h>
#include "defafd.h" /* définition de l'automate */

/** reconnaît un mot suivi de EOF sur l'entrée standard
 *@return 0 si non reconnu, 1 sinon
 */
int accepter(){
  int  etat=EINIT; /* unique état initial */
  int c; /* caractère courant */
  while ((c=getchar())!=EOF) /* Tq non fin de fichier */
    if (TRANS[etat][c]!=-1) /* si transition définie */
      etat=TRANS[etat][c]; /* Avancer */
    else return 0; /* sinon Echec de reconnaissance */
  return FINAL[etat]; /* OK si dans un état final */
}
int main(){ /* Programme principal */
  creerAfd(); /* Construction de l'AFD */
  printf("Saisissez un mot matchant a(b+c)?|bd suivi de EOF (CTRL-D) SVP : ");
  if (accepter())
    printf("\nMot reconnu !\n");
  else
    printf("\nMot non reconnu !\n");
  return 0;
}
```

Si l'on compile ce programme C et qu'on l'exécute, on obtient les résultats suivants :

```
> accepter
Saisissez un mot matchant a(b+c)?|bd suivi de EOF (CTRL-D) SVP : abbbc
Mot reconnu !
```

```
> accepter
Saisissez un mot matchant a(b+c)?|bd suivi de EOF (CTRL-D) SVP : abd
Mot non reconnu !
```

Il existe d'autres types d'implémentation de la table de transition d'un AFD :
— par un multigraphe étiqueté chaîné (pointeurs),
— par une table de transition plus petite ; la taille de la table est alors : $taille(TRANS) = |E| * |V|$. Cette solution est adoptée par le programme lex (voir section 2.5), avec une structure de données réduisant la taille de la table qui est souvent "creuse".

2.3 Analyseur lexical

L'analyse lexicale est bien plus complexe que la simple reconnaissance d'un mot.
— Suite à la reconnaissance d'un mot ou **lexème**, l'analyseur lexical doit retourner un **jeton** entier associé à la **catégorie lexicale** du mot accepté. Un jeton (*token*) est généralement représenté par un entier positif. Les entiers inférieurs à 256 sont réservés aux **mots clés** composés d'un seul caractère : ("{", ";", "]", ...). Leur code (ASCII, ISO Latin1, ...) correspondra ainsi à leur jeton. Chaque mot clé de plus d'une lettre est également associé à son jeton : (if, 300), (else, 301), (while, 302), ... On définira également un jeton pour chaque catégorie lexicale variable : (littéral entier, 303), (littéral chaîne, 304), ... Pour les catégories lexicales variables, il faudra également "retourner" une **valeur sémantique** associée. Par exemple, pour les littéraux entiers on pourrait retourner la valeur entière correspondante, pour les identificateurs le lexème lui-même ou l'indice d'entrée correspondant dans la table des symboles.
— De plus, un analyseur lexical doit reconnaître une **suite** de lexèmes dans un **flot** de caractères. Dans l'automate d'états finis déterministe (AFD), chaque état terminal est associé à un jeton retournable. C'est le chemin parcouru dans l'automate qui déterminera le jeton à retourner. Cela peut poser problème lorsque un mot du langage est préfixe d'un autre. Lorsqu'on est sur le dernier caractère du préfixe, pour savoir quel jeton retourner, il est nécessaire de regarder le caractère suivant : si celui-ci étend le lexème reconnu, on le lira et on avancera dans l'automate (**règle du mot le plus grand possible**), sinon on reconnaîtra le préfixe. Par exemple, `while(` est reconnu comme un mot clé puis une parenthèse, alors que `while1` est reconnu comme un identificateur. Attention, si on a avancé dans l'AFD et que l'on se retrouve dans un état non terminal sans pouvoir avancer, il faudra **reculer** afin de retourner dans le dernier état terminal parcouru ! Ce recul nécessite de rejeter dans le flot d'entrée (**ungetc**) les caractères qui ont été lus en trop. En reprenant l'exemple 4 précédent, le mot "abd" doit être analysé comme une suite des jetons A, BD même si à un moment l'analyseur avait avancé jusqu'à l'état EAB. Enfin, une convention habituelle permet de retourner le jeton 0 lorsqu'on est arrivé à la fin du flot.
— Enfin, l'analyseur lexical doit **filtrer** un certain nombre de mots inutiles pour l'analyseur syntaxique (blancs (espace, tabulations, retour à la ligne), commentaires, ...).

Prenons l'exemple du morceau de code correspondant à la fonction `main()` du fichier `accepter.c` précédent et voyons la suite de couple (jeton, valeur sémantique) que doit successivement retourner la fonction d'analyse lexicale du compilateur C :

```
(VOID,) (ID,'main') ('(',) (')',) ('{',) (ID,'creerAfd') ('(',) (')',) (';',)
(ID,'printf') ('(',) (LITTERALCHAINE,'Saisis...') (')',) (';',)
(IF,) ('(',) (ID,'accepter') ('(',) (')',) (')',)
(ID,'printf') ('(',) (LITTERALCHAINE,'\nMot...') (')',) (';',)
(ELSE,) (ID,'printf') ('(',) (LITTERALCHAINE,'\nMot...') (')',) (';',)
(RETURN,) (LITTERALENTIER,0) (';',) ('}')
```

L'algorithme 2 en page 8 décrit le fonctionnement d'un tel analyseur lexical.
Quelques remarques sur cet algorithme 2 :
— la gestion des mots non reconnus est la suivante : retourner le jeton correspondant au code ASCII du premier caractère. Contrairement à cela, Lex lui ne retourne aucun jeton mais envoie ce premier caractère sur la sortie standard et tenter de se resynchroniser sur le caractère suivant ;
— on suppose dans cet algorithme que le symbole $ est retourné à l'infini par carSuivant() lorsqu'on est parvenu à la fin du flot ;
— Remarquons que dans le cas où l'état initial est également final, le mot vide est donc acceptable. Par conséquent, sur un mot non acceptable ou sur le mot vide, l'analyseur lexical retournera une suite infinie de jetons associés à l'état initial !
— le caractère minimal d'un AFD n'est pas une bonne propriété pour les analyseurs lexicaux dans la mesure ou la minimisation d'un AFD fusionne plusieurs états terminaux ce qui interdit le retour de jetons distincts. Il suffit de construire l'AFDM du langage $\{< b >, < /b >\}$ pour s'en persuader !
— cet algorithme ne gère pas le filtrage (suppression des lexèmes inutiles (blancs, commentaires)).

Algorithme 2 : Analyseur lexical

Données : $B = (V, E, D = \{d\}, A, T)$ un AFD ; $JETON[A]$ le tableau des jetons associés à chaque état final ; $flot$ un flot de caractères terminé par \$
Résultat : (Entier : le jeton reconnu, Chaîne : le lexème reconnu)
Fonction `analex(`*B, JETON[A], flot*`)` : (Entier, Chaîne);
début
  // Initialisations;
  etat=d;
  lexeme="";
  efinal=-1;
  lfinal=0;
  tant que *((c=carSuivant(flot))≠ $) et* $(etat, c, e) \in T$ **faire**
    lexeme=lexeme . c;
    etat=e;
    si $e \in A$ **alors**
      efinal=e;
      lfinal=|lexeme|;
  si $etat \in A$ **alors**
    rejeter(flot, c);
    retourner *(*$JETON[etat]$*,lexeme)*;
  sinon
    si $efinal > -1$ **alors**
      rejeter(flot, c);
      rejeter(flot, sous-chaine(lexeme,lfinal,|lexeme|));
      retourner *(*$JETON[efinal], lexeme[0, lfinal - 1]$*)*;
    sinon
      // pas d'état final;
      si *lexeme="" et c=$* **alors**
        retourner *(0,"")*;
      sinon
        si *lexeme=""* **alors**
          retourner *(c,c)*;
        sinon
          rejeter(flot, c);
          rejeter(flot, sous-chaine(lexeme,1,|lexeme|));
          // tout sauf le 1er car;
          retourner *(lexeme[0], lexeme[0])*;

2.4 Implémentation des analyseurs lexicaux

L'implémentation d'une fonction d'analyse lexicale `int analex()` et de l'AFD de l'exemple 4 est décrite dans l'exemple 5. Quelques remarques :

- En C, une seule valeur pouvant être retournée par une fonction, on choisit de retourner le jeton entier et d'implémenter la valeur sémantique dans une variable globale `lexeme` de type chaîne ;
- le **filtrage** des séparateurs (blancs : espaces, tabulations, ...) et des commentaires est réalisée par un jeton négatif affecté aux états finaux à filtrer. Il suffit alors de modifier le retour d'un jeton négatif en appel récursif à analex() : `return JETON[etat];` devient alors `return (JETON[etat]<0 ? analex() : JETON[etat]);`. On trouvera ces changements dans le fichier `analex.h`.

Exemple 5
Soit l'AFD de l'exemple 4. On transforme la définition de l'automate pour ajouter la définition des jetons dans un tableau entier JETON remplaçant le tableau FINAL :

```
JETON[EA]=300;JETON[EABC]=301;JETON[EBD]=-302; /* jetons des états finaux */
```

Remarquons, que les lexèmes "bd" seront filtrés car le jeton correspondant est négatif ! Nous représentons la fonction d'analyse lexicale `int analex()` dans le fichier `analex.h` suivant :

```
/**@file analex.h
 *@author Michel Meynard
 *@brief Définition de la fon  analex
 */
char lexeme[1024]; /* lexème courant de taille maxi 1024 */

/** reconnaît un mot (lexème) sur l'entrée standard et retourne un jeton
 * correspondant à la catégorie lexicale du lexème.
 * Le filtrage est permis gâce aux jetons négatifs.
 *@return un entier négatif si erreur, positif si OK, 0 si fin de fichier
 */
int analex(){
  int  etat=EINIT; /* unique état initial */
  int efinal=-1; /* pas d'état final déjà vu */
  int lfinal=0; /* longueur du lexème final */
  int c;char sc[2];int i; /* caractère courant */
  lexeme[0]='\0'; /* lexeme en var globale (pour le main)*/
  while ((c=getchar())!=EOF && TRANS[etat][c]!=-1){ /* Tq on peut avancer */
    sprintf(sc,"%c",c); /* transforme le char c en chaine sc */
    strcat(lexeme,sc); /* concaténation */
    etat=TRANS[etat][c]; /* Avancer */
    if (JETON[etat]){ /* si état final */
      efinal=etat; /* s'en souvenir */
      lfinal=strlen(lexeme); /* longueur du lexeme egalement */
    } /* fin si */
  } /* fin while */
  if (JETON[etat]){ /* état final */
    ungetc(c,stdin); /* rejeter le car non utilisé */
    return (JETON[etat]<0 ? analex() : JETON[etat]);/* ret le jeton ou boucle*/
  }
  else if (efinal>-1){ /* on en avait vu 1 */
    ungetc(c,stdin); /* rejeter le car non utilisé */
    for(i=strlen(lexeme)-1;i>=lfinal;i--)
      ungetc(lexeme[i],stdin); /* rejeter les car en trop */
    lexeme[lfinal]='\0'; /* voici le lexeme reconnu */
    return (JETON[efinal]<0 ? analex() : JETON[efinal]);/* ret jeton ou boucle*/
  }
  else if (strlen(lexeme)==0 && c==EOF)
    return 0; /* cas particulier */
  else if (strlen(lexeme)==0){
    lexeme[0]=c;lexeme[1]='\0'; /* retourner (c,c) */
    return c;
  }
```

```
  else {
    ungetc(c,stdin); /* rejeter le car non utilisé */
    for(i=strlen(lexeme)-1;i>=1;i--)
      ungetc(lexeme[i],stdin); /* rejeter les car en trop */
    return lexeme[0];
  }
}
```

Enfin la fonction principale est codé dans le programme C suivant :

```
/**@file analex.c
 *@author Michel Meynard
 *@brief Prog principal appelant itérativement analex()
 */
#include <stdio.h>
#include <string.h>
#include "afd.h" /* Définition de l'AFD et des JETONS */
#include "analex.h" /* Définition de la fon : int analex() */

int main(){
  int j; /* jeton retourné par analex() */
  char *invite="Saisissez un(des) mot(s) matchant a(b+c)?|bd suivi de EOF (CTRL-D) SVP : ";
  creerAfd(); /* Construction de l'AFD à jeton */
  printf("%s",invite); /* prompt */
  while((j=analex())!=0){ /* analyser tq pas jeton 0 */
    printf("\nRésultat : Jeton = %d ; Lexeme = %s\n%s",j,lexeme,invite);
  }
  return 0;
}
```

Si l'on compile ce programme C et qu'on l'exécute, on obtient les résultats suivants.

```
Saisissez un(des) mot(s) matchant a(b+c)?|bd suivi de EOF (CTRL-D) SVP : abdbdabbc
Résultat : Jeton = 300 ; Lexeme = a
Résultat : Jeton = 301 ; Lexeme = abbc
Saisissez un(des) mot(s) matchant a(b+c)?|bd suivi de EOF (CTRL-D) SVP : 0ab
Résultat : Jeton = 48 ; Lexeme = 0
Résultat : Jeton = 300 ; Lexeme = a
Résultat : Jeton = 98 ; Lexeme = b
Résultat : Jeton = 10 ; Lexeme =
```

Remarquons que sur l'entrée standard Unix le CTRL-D tapé en début de ligne génère un EOF, mais après une chaîne de caractères, le CTRL-D (parfois doublé à cause des ungetc) génère un vidage (flush) du tampon d'entrée sans caractère supplémentaire à la différence du ENTREE qui envoie un retour à la ligne (`'\n'` codé par 10).

Pour conclure, avec un langage réel de taille importante, il devient difficile de construire manuellement l'AFD sans se tromper (plusieurs centaines de transitions). De plus, l'évolution permanente de la grammaire d'un langage en cours de conception rend nécessaire l'utilisation d'un outil informatique pour modéliser le langage lexical à l'aide d'expressions régulières. L'outil aura comme mission de transformer ces expressions en AFD à jeton et de fournir une fonction d'analyse lexicale.

2.5 Un langage et un outil pour l'analyse lexicale : (f)lex

Pour plus d'informations sur flex, faire `man flex`. Lex est un outil permettant de générer un programme d'analyse lexicale à partir de définitions régulières de modèles (expressions régulières) et d'actions à exécuter lors de la reconnaissance de ces modèles. Il existe différentes versions de lex (lex, flex, pclex,...) sur différentes plateformes et permettant l'utilisation de différents langages d'actions (C, ada, ...). Les plus usuelles tournent sous Unix et utilisent le C. Nous utiliserons "Flex" qui est une version gratuite, rapide, n'ayant pas besoin de bibliothèque. Ce logiciel peut facilement être téléchargé depuis Internet.

2.5.1 Un exemple

Exemple 6
Analyseur lexical de l'exemple 4 réécrit en lex :

```
/* ZONE DE DEFINITION  (OPTIONNELLE) */

/* evite la definition de yywrap() */
%option noyywrap

/* ZONE DES REGLES apres le double pourcent (OBLIGATOIRE) */
%%
a  {return 300; /* ret un jeton */}
ab+c  {return 301; /* ret un jeton */}
bd {/* ne rien faire : filtrer */}
.|\n {return yytext[0]; /* ret le code ASCII pour tout le reste */}
%%
/* ZONE DES FONCTIONS C (OPTIONNELLE) */
int main(){
  int j; char *invite="Saisissez un(des) mot(s) matchant a(b+c)?|bd; finissez par EOF (CTRL-D) SVP : ";
  printf(invite);
  while ((j=yylex())!=0)
    printf("\nJeton : %i; de lexeme %s\n%s",j,yytext,invite);
  return 0; /* 0 en fin de fichier */
}
```

Après compilation flex, `flex analflex.l`*, puis compilation C* `gcc -o analflex lex.yy.c`*, il ne reste plus qu'à lancer l'exécutable* `analflex` *obtenu :*

```
Saisissez ... : abbbbcbdbdabdabbc
Jeton : 301; de lexeme abbbbc
Jeton : 300; de lexeme a
Jeton : 301; de lexeme abbc
<CTRL>-<D>
```

L'analyseur lexical généré tente, de manière itérative, de reconnaitre une expression régulière (pattern matching) puis exécute les instructions C correspondantes. L'analyseur termine sur la fin de fichier (EOF) de l'entrée standard (CTRL-D pour le terminal). Les mots ne correspondant à aucune expression régulière **sont rejetés dans la sortie standard** sans aucun traitement particulier. C'est pourquoi la dernière règle `.|\n` a été ajoutée : par défaut, tout cartactère non reconnu est retourné.

Au cœur du source C `lex.yy.c` généré par flex, la fonction C : `int yylex()` d'analyse lexicale permet de retourner un jeton entier correspondant au modèle reconnu. Dans l'exemple précédent, la fonction principale : `int main()` appelle `yylex()` itérativement jusqu'au caractère de fin de fichier. La résolution de l'ambiguïté de reconnaissance est obtenue d'une part, par la tentative de toujours reconnaitre le mot le plus long possible, d'autre part par l'ordre des expressions régulières dans le source lex.

Si l'on observe le code C généré dans `lex.yy.c`, on s'aperçoit que l'automate fini déterministe calculé par flex est codé dans un tableau statique du programme C.

2.5.2 Syntaxe et sémantique des sources Flex

Architecture

Un source lex comprend 3 parties séquentielles :

- une partie optionelle de définitions. Elle contient :
  - les directives d'inclusion et de définition globales C (variables, types, ...) entre `%{` et `%}` ;
  - les définitions spécifiques à flex (abbréviations, start condition, options) ;
- Une partie obligatoire de règles lex délimitée par %% au début. C'est la partie centrale du source lex qui définit l'analyseur lexical en associant des instructions C à des expressions régulières.
- Une partie optionelle de fonctions C définies par l'utilisateur délimitée par %% au début. C'est là que l'on peut définir le main().

Les règles lex

Une règle lex se présente de la façon suivante : une expression régulière, suivie de séparateur(s), suivie de

- d'un bloc d'instructions C ou C++ encadré par des accolades ou bien
- " ;" (ne rien faire) ou bien
- d'une instruction C à exécuter.

L'espace et la tabulation sont les séparateurs qui divise la règle en deux. Le modèle lexical doit commencer en début de ligne et la règle doit se terminer par un " ;" ou une fin de bloc C "}".

Représentation des expressions régulières

Soit e et r deux expressions régulières quelconques, c et d deux caractères, m et n deux nombres entiers positifs, le tableau suivant indique les opérateurs utilisés par lex.

Exemple	Signification	Opérateur(s)
abc	concaténation implicite	er
Monsieur|Madame	union	e|r
b*	opération * : 0 à n 'b'	e*
b+	opération + : 1 à n 'b'	e+
cartons ?	optionnel : carton ou cartons	e ?
(abc)*	parenthésage pour priorités	(e)
`[ace]`	classe de car : 1 caractère parmi	`[dc]`
`[a-z][0-9]`	1 caractère minuscule suivi d'un chiffre	`[c-d]`
`[^abc]`	1 caractère sauf a, b ou c	`[^cd]`
`"[*+)("`	évite l'interprétation des opérateurs	`"dc"`
`\*`	le caractère * (et non pas l'opérateur)	`\c`
.	un car quelconque hormis newline	.
`[0-9]{3}`	trois chiffres	e{n}
a{1,10}	entre 1 à 10 'a' contigus	e{m,n}
a{3,}	au moins 3 'a' contigus	e{m,}
`^Bonjour`	Bonjour en début de ligne	`^e`
Au revoir$	Au revoir en fin de ligne (pas en EOF)	e$
`^Bonjour, Au revoir$`	interdit (1 seul opérateur contextuel)	
Bonjour/(Monsieur)	Bonjour seulement si suivi par Monsieur	e/r
`<etat1>Dupont`	Dupont seulement si on est dans l'état etat1	`<state>e`
`<<EOF>>`	fin de fichier (seulement dans flex)	`<<EOF>>`
`<state><<EOF>>`	fin de fichier dans un certain état (seulement dans flex)	`<state><<EOF>>`
`{chiffre}`	chiffre est une définition (alias) dans la 1ère partie du source lex	`{def}`

Instruction(s) C

La partie droite de chaque règle correspond à une suite de longueur quelconque d'instructions C. Le texte inclu entre accolades sera recopié intégralement dans `lex.yy.c` sans aucune analyse ni modification. Il doit donc correspondre à un source C correct.

Les instructions C peuvent faire appel à des fonctions prédéfinies par lex ou définies par l'utilisateur dans la troisième partie du source lex. En particulier, avec `flex`, on peut ne pas utiliser la librairie `flex libfl.a` à condition de définir la fonction principale `main()` ainsi que la fonction `int yywrap()`. Par exemple, pour éviter l'édition de liens avec la librairie flex, on pourra simplement écrire dans la troisième partie :

```
int yywrap() {return 1;} /* pas d'enchaînement sur un autre fichier */
main() {while (yylex()!=0) {} } /* boucle sans rien faire jusqu'à eof */
```

Les instructions C peuvent référerencer une variable :
- soit prédéfinie par lex : la chaîne `char* yytext` de longueur `int yyleng` correspond au mot reconnu dans le texte à analyser (lexème) ;
- soit définie en partie définitions : dans ce cas, la variable est globale ;
- soit définie juste après l'accolade : dans ce cas, la variable est locale à la règle.

Exemple 7
Le source lex suivant illustre l'utilisation des variables :

```
%{ int glob=0; %}
%%
-?[1-9]+ {int loc=5; glob++;loc++;
          printf("%d ème entier de taille %d; loc= %d",glob,yyleng,loc);
        }
```

Une exécution de ce programme donne :

```
12
1 ème entier de taille 2; loc= 6
123
2 ème entier de taille 3; loc= 6
-1
3 ème entier de taille 2; loc= 6
```

Variables prédéfinies

yytext chaîne de car (`char *`) contenant le lexème en cours de reconnaissance ;
yyleng longueur (`int`) de `yytext` ;
yyin flot d'entrée des caractères de type `FILE*` (par défaut `stdin`) ; On peut rediriger le flot d'entrée sur le premier argument du main en faisant : `yyin=fopen(argv[1],"r");`
yyout sortie standard de type `FILE*`. Pour y afficher, faire : `fprintf(yyout, "...");`

Fonctions prédéfinies

int yylex() lit un lexème depuis le flot d'entrée et retourne le jeton associé. Retourne le jeton 0 pour finir.
int input() lecture d'un caractère depuis le flot d'entrée (`yyinput` en C++) ; `input()` équivaut à `fgetc(yyin)` ;
void unput(int) retour dans le flot d'entrée d'un car ; `unput(c)` équivaut à `ungetc(c,yyin)` ;
int yywrap() lorsque l'analyseur `yylex()` arrive en fin de fichier (EOF), il appelle `yywrap()`. Si `yywrap` retourne 1 (par défaut) alors `yylex()` retourne 0 (fin d'analyse). Si on voulait enchaîner sur un autre fichier, il faut redéfinir dans la partie "définitions" du source lex, la fonction `yywrap()` afin qu'elle fasse pointer `yyin` sur le nouveau fichier puis retourne 0 ;
yymore() concatène dans `yytext` le prochain lexème avec celui en cours ;
yyless(int n) replace le lexème reconnu `yytext` dans le flot d'entrée à l'exception de ses n premiers caractères ;
ECHO affiche `yytext` ; `ECHO` équivaut à `fprintf(yyout,yytext)` ;
REJECT rejette le lexème reconnu dans le flot d'entrée et s'interdit de reconnaitre la règle courante au prochain essai (appel de `yylex()`).
BEGIN(etat) positionne l'automate dans la condition de départ `etat`. Cet état doit avoir été défini dans la première partie grâce à `%Start etat` ou à `%x etat`. `BEGIN(0)` permet de revenir à l'état normal.
int main() par défaut, la librairie de `lex` (`libl.a`) ou de `flex` (`libfl.a`) définissent une fonction pricipale qui appelle `yylex()` jusqu'à ce que celle-ci retourne 0.

Ambiguités de correspondance

Règle de la plus longue correspondance (match) si un préfixe (début de chaîne) correspond à plusieurs expressions régulières possibles, `lex` choisira l'expression régulière correspondant à la plus longue extension. Par exemple, avec les règles suivantes :

```
end       {return 300;}
[a-z]+    {return 301;}
```

Le mot `endemique` se verra appliquer la seconde règle (identificateur) et `yylex()` retournera 301.

Attention aux opérateurs contextuels en avant qui comptabilisent les caractères en avant : par exemple, l'expression régulière `a$` sera préféré à l'expression `a` pout tout a en fin de ligne.

Règle du premier trouvé si la longueur de correspondance est égale pour plusieurs règles, alors c'est la première dans la liste qui est déclenchée. Dans l'exemple précédent, le mot `end` déclenchera le retour de 300. Par conséquent, pour un langage donné, il faut toujours placer les règles concernant les mots-clés au début.

Attention aux opérateurs contextuels qui provoquent parfois des "erreurs" ! En effet, l'utilisation des 2 règles suivantes provoque un conflit gagné par la première règle (à l'encontre de la règle du plus long lexème) :

```
a+$     {return 300; /* ret un jeton */}
^a+\n   {return 301; /* ret un jeton */}
```

En inversant l'ordre de ces deux règles, tout se passe cependant comme prévu. En fait, les opérateurs contextuels de suffixe (`$, /`) sont consommés après le lexème et c'est ce mot qui doit être considéré comme le plus long possible. Ensuite, le suffixe sera rejeté dans `yyin`.

Partie définitions

Il existe différentes sortes de définitions :

Définitions C toute ligne de la partie définitions débutant par un espace ou une tabulation est recopiée au début du source C généré par lex. Ces lignes seront donc externes à toute fonction C du code correspondant à l'automate. Il en va exactement de même pour tout ce qui est inclus entre `%{` et `%}` **seuls et en début de ligne**, ces délimiteurs étant détruits dans `lex.yy.c`. A part les variables globales, cette partie permet d'inclure des macros `#include #define`, des `typedef`, ...

Abbréviation de modèle certaines parties de modèles revenant fréquemment dans les règles, on peut en définir des alias selon la syntaxe suivante : `nomAlias séparateur(s) modèle`. Par exemple :

```
chiffre   ([0-9])
minuscule ([a-z])
exposant  ([DEde][-+]?{chiffre}+)
```

Dans cet exemple, chiffre désigne l'alias de `[0-9]`. Ces alias seront principalement utilisés dans les expressions régulières en les entourant d'accolades. Le parenthésage sera utilisé systématiquement pour éviter des problèmes liés aux priorités.

Start Condition permet de conditionner la reconnaissance de certaines expressions régulières selon la condition dans lequel l'analyseur se trouve. Par exemple, `%x DANSCHAINE DANSCOMMENT` définit deux conditions exclusives. Celles-ci pourront être utilisés en préfixe des expressions régulières comme dans l'exemple suivant :

```
["]               {BEGIN(DANSCHAINE);}
<DANSCHAINE>[^"]+ {yylval.s=strdup(yytext);}
<DANSCHAINE>["]   {BEGIN(INITIAL); return LITCHAINE;}
```

Au départ la condition initiale s'appelle INITIAL et vaut 0. Lorsque flex reconnaît le guillemet, il passse dans la condition DANSCHAINE où il va reconnaître l'intérieur de la chaîne. Après avoir reconnu le guillemet final, il retournera le jeton de littéral chaîne.

La définition des états peut également se faire par `%s s1 s2` (inclusif). La différence entre les conditions inclusives et exclusives réside dans le fait que dans le cas inclusif, les règles préfixées de condition sont prioritaires mais les autres règles (sans conditions) seront utilisées s'il n'y a pas de correspondance possible ! Il est souvent préférable d'utiliser l'exclusivité `%x s1 s2`.

Options flex commençant toujours par le mot `%option` telles que `%option noyywrap` : un seul fichier, `%option yylineno` : numéro de ligne, ...

Troisième partie

Cette partie permet d'écrire des fonctions C utilisées dans les parties droites des règles. On peut également redéfinir les fonctions `main(), yywrap(), input(), unput(char), ...` afin de surcharger leur version `flex`. Ces fonctions peuvent également être redéfinies dans un fichier inclus.

2.5.3 La commande flex

Principales options de la commande `flex` :

flex -d débogue un source flex en affichant lors de l'exécution la règle reconnue (ligne) et le lexème ;

flex -T trace l'automate construit en donnant : l'AFN (nfa), l'AFD (dfa), et les classes de caractères définies ;

flex -v (verbose) donne des informations statistiques sur l'automate généré ;

flex -s supprime la règle par défaut qui consiste à envoyer sur la sortie standard tout caractère non reconnu.

makefile

Voici la partie du `makefile` correspondant à la génération d'applications à partir de source `flex` d'extension `.l`. Si l'on veut utiliser `flex` sans sa bibliothèque (extension `.fl`), il suffit de définir les fonctions **int main()** et **int yywrap()**.

```
.SUFFIXES:.fl
CC=gcc
CFLAGS=-g
LEX=flex
LEXLIBRARY=-lfl
.l:     # avec la librairie LEX
        @echo debut $(LEX)-compil : $<
        $(LEX) $<
        @echo debut compil c avec edition de liens de lex.yy.c
        $(CC) $(CFLAGS) -o $* lex.yy.c $(LEXLIBRARY)
        @echo fin $(LEX)-compil de : $<
        @echo Vous pouvez executer : $*
.fl:    # sans librairie (seulement flex) -> main et yywrap
        @echo debut flex-compil : $<
        flex $<
        @echo debut compil c avec edition de liens de lex.yy.c
```

```
        $(CC) $(CFLAGS) -o $* lex.yy.c
        @echo fin flex-compil de : $<
        @echo Vous pouvez executer : $*
```

2.5.4 Actions C++

Il est possible d'utiliser `flex` avec des actions en C++. Il suffit alors de compiler `lex.yy.c` avec un compilateur C++. Soit le source `flex` suivant :

```
%{
 #include <iostream.h>
 class A{
 public:
  void essai(){cout<<"Identif ";
  }
 };
%}
%%
[a-z]([a-z]|[0-9])*     {return 4;}
.                       {return 5;}
%%
int main(){
  A a; int i;
  while ((i=yylex())!=0)
    if (i==4) a.essai();
}
```

Après compilation par `flex exempleC++.l+` puis `g++ -g -o exempleC++ lex.yy.c -lfl`, on obtient un exécutable.

makefile pour le C++

Voici les 2 entrées de `makefile` pour les sources `flex` contenant des instructions C++ :

```
CPP=g++
CPPFLAGS=-g
.l+:    # C++ avec la librairie LEX
        $(LEX) $<
        $(CPP) $(CPPFLAGS) -o $* lex.yy.c $(LEXLIBRARY)
.fl+:   # C++ sans la librairie LEX
        $(LEX) $<
        $(CPP) $(CPPFLAGS) -o $* lex.yy.c
```

2.5.5 Liaison avec un analyseur syntaxique

Lorqu'il est utilisé avec un analyseur syntaxique généré par `yacc` ou `bison`, c'est la fonction d'analyse syntaxique `yyparse()` qui appelle itérativement `yylex()` pour obtenir les jetons correspondants au fichier analysé. La fonction principale `int main()` appelle alors `yyparse()`. Une ou plusieurs variables globales, `yylval` par exemple, peuvent être alors partagées par les 2 fonctions `yylex()` et `yyparse()`.

2.6 Algorithmique

Nous allons étudier les différents algorithmes utilisés par Flex pour construire "l'automate" déterministe codé en C.

2.6.1 Traduction des expressions régulières

On utilise la construction de "Thompson" qui admet des AFN possédant des ε-transitions mais ayant un unique état initial et un unique état final. La donnée est constituée d'une expression régulière r (sans $\emptyset$) sur l'alphabet V. Le résultat est un AFN construit par l'algorithme 3. Le principe de celui-ci revient à associer récursivement un automate à chaque noeud de l'arbre syntaxique de l'expression régulière.

Quelques propriétés de l'algorithme 3 :

— Correction : l'AF construit reconnaît le langage L(r) défini par l'expression régulière r.

Algorithme 3 : construction d'un automate équivalent à une expression régulière

Données : r une expression régulière sur V
Résultat : $B = (V, E, D, A, T)$
1 Construire l'arbre a de construction inductive de r // *arbre syntaxique de r*;
2 i=0 // *numéro d'état*;
3 B=`arbreVersAF(a)` // *appel à la fonction définie dans l'algorithme 4* ;

Algorithme 4 : construction d'un automate à partir d'un arbre

Données : a un arbre syntaxique d'une expression régulière r
Résultat : $B = (V, E, D, A, T)$
Fonction `arbreVersAF(a)` : automate;
si *a est une feuille étiquetée par un symbole* $s \in V \cup \{\varepsilon\}$ **alors**
  $B = (V, \{i, i+1\}, \{i\}, \{i+1\}, \{(i, s, i+1)\})$;
  i=i+2;
  retourner B;
si *a est étiquetée par* $\bullet$ **alors**
  B_g =`arbreVersAF(`*sous-arbre-gauche(a)*`)` // $(V, E_g, \{d_g\}, \{a_g\}, T_g)$;
  B_d =`arbreVersAF(`*sous-arbre-droit(a)*`)` // $(V, E_d, \{d_d\}, \{a_d\}, T_d)$;
  retourner $B = (V, E_g \cup E_d, \{d_g\}, \{a_d\}, T_g \cup T_d \cup \{a_g \varepsilon d_d\})$ // *l'état final de* B_g *est "fusionné" à l'état initial de* B_d;
si *a est étiquetée par* $|$ **alors**
  B_g =`arbreVersAF(`*sous-arbre-gauche(a)*`)` // $(V, E_g, \{d_g\}, \{a_g\}, T_g)$;
  B_d =`arbreVersAF(`*sous-arbre-droit(a)*`)` // $(V, E_d, \{d_d\}, \{a_d\}, T_d)$;
  $B = (V, E_g \cup E_d \cup \{i, i+1\}, \{i\}, \{i+1\}, T_g \cup T_d \cup \{i\varepsilon d_g, i\varepsilon d_d, a_g \varepsilon i+1, a_d \varepsilon i+1\})$ // *on parallèlise* B_g *et* B_d;
  i=i+2;
  retourner B;
si *a est étiquetée par* $*$ **alors**
  B_g =`arbreVersAF(`*sous-arbre(a)*`)` // $(V, E_g, \{d_g\}, \{a_g\}, T_g)$;
  $B = (V, E_g \cup \{i, i+1\}, \{i\}, \{i+1\}, T_g \cup \{i\varepsilon d_g, i\varepsilon i+1, a_g \varepsilon i+1, a_g \varepsilon d_g\})$ // *on crèe un circuit sur* B_g;
  i=i+2;
  retourner B;

— L'AF construit a au plus deux fois plus d'états que $|r|$.
— L'AF construit a un état initial et un état final.
— Chaque état (non final) possède, soit 1 ou 2 ε-transitions sortantes, soit une transition sortante étiquetée par un symbole de V.
— Chaque état (non initial) possède, soit 1 ou 2 ε-transitions entrantes, soit une transition entrante étiquetée par un symbole de V.
— L'état final n'a pas de transition sortante, l'état initial n'a pas de transition entrante.

Les preuves de ces propriétés sont réalisées par l'analyse de la fonction récursive `arbreVersAF`.

La difficulté de mise en oeuvre de cet algorithme réside dans la construction de l'arbre de dérivation. En effet, la grammaire des expressions régulière est algébrique non rationnelle. Une programmation récursive ad hoc permet cependant de le réaliser. Il ne reste plus ensuite qu'à déterminiser l'AF ainsi construit pour construire un AFD équivalent à une expression régulière.

2.6.2 Déterminisation

On va écrire l'algorithme 5 de déterminisation d'un AFN $N = (V, E, D, A, T)$; l'idée consiste à fusionner l'ensemble des états où l'AFN peut être à un "instant" donné en un seul état de l'AFD $D = (V, DE, \{d\}, DA, DT)$. Pour cela, un état de DE sera modélisé dans l'algo. par un ensemble d'états de E. Il reste à la fin de l'algorithme 5 à numéroter ces ensembles. L'Epsilon fermeture d'un ensemble d'états consiste à effectuer la fermeture réflexo-transitive par des epsilon transitions depuis ces états.

A tout chemin menant d'un état initial à un état final de N, donc à tout mot de L(N), correspond un chemin de d à un état final dans D. De plus, pour un chemin menant à un état final, l'état $\{\dots e_{n+1} \dots\}$ est final (Voir dans l'algorithme 5 : $DA = \{Y \in DE / Y \cap A \neq \emptyset\}$).

Remarquons que cette déterminisation permet de supprimer tous les chemins inaccessibles.

Algorithme 5 : déterminisation d'un automate

Données : $N = (V, E, D, A, T)$
Résultat : $B = (V, DE, \{d\}, DA, DT)$
d=EpsilonFermeture(D) ; // *on initialise l'ensemble des états initiaux comme unique état de départ non marqué*;
DE={d};
tant que *il existe un état* $G = \{e_1, e_2, \ldots, e_n\}$ *non marqué dans DE* **faire**
  marquer G // *on traite une seule fois chaque état de l'AFD B*;
  pour chaque $x \in V$ **faire**
    $X = EpsilonFermeture(\bigcup_{i=1}^{n}\{e_j\})$ tel que $e_i \in G$ et $(e_i x e_j) \in T$ // *X est l'ensemble des états atteignables par x à partir de G*;
    si $X \neq \emptyset$ **alors**
      $DE = DE \cup \{X\}$;
      $DT = DT \cup \{(GxX)\}$ // *ajouter la transition dans l'AFD*;
$DA = \{Y \in DE / Y \cap A \neq \emptyset\}$ // *les états finaux de B sont ceux qui contiennent au moins un état final de N*;
numéroter les états de DE et substituer ces numéros dans DE,DA,DT ;

Exemple 8
Déterminisons l'AFN N suivant : $N = \{\{a, b\}, \{1..4\}, \{1, 2\}, \{3, 4\}, \{1a3, 1a4, 2a3, 2b4\}\}$
traçons l'algorithme :
$DE = \{\{1, 2\}*\}$;
$x = a; X = \{3, 4\}; DE = \{\{1, 2\}*, \{3, 4\}\}; DT = \{(\{1, 2\}a\{3, 4\})\}$
$x = b; X = \{4\}; DE = \{\{1, 2\}*, \{3, 4\}, \{4\}\}; DT = \{(\{1, 2\}a\{3, 4\}), (\{1, 2\}b\{4\})\}$
$DE = \{\{1, 2\}*, \{3, 4\}*, \{4\}\}$;
$x = a$ *puis* $b; X = \emptyset$
$DE = \{\{1, 2\}*, \{3, 4\}*, \{4\}*\}$;
$x = a$ *puis* $b; X = \emptyset$
$DA = \{\{3, 4\}, \{4\}\}$
numérotation : $\{1, 2\} \to 1; \{3, 4\} \to 2; \{4\} \to 3; D = \{\{a, b\}, \{1..3\}, \{1\}, \{2, 3\}, \{1a2, 1b3\}\}$.

2.6.3 Minimisation

Rappelons que la forme canonique d'un langage régulier est son AFD minimal. Etudions l'algorithme 6 de minimisation d'un AFD $B = (V, E, \{d\}, A, T)$. On suppose en entrée un AFD **complet** en ajoutant si nécessaire un état puits. On va construire incrémentalement une suite de partitions P_i, composées de classes d'états. On dit que 2 états i, j d'une même classe C sont distinguables par un symbole $x \in V$ ssi la reconnaissance de x n'aboutit pas pour ces deux états à la même classe de la partition courante. On va partitionner les états de l'automate en classes d'états distinguables les unes par rapport aux autres puis ces classes représenteront les états du nouvel AFD Minimal M.

Algorithme 6 : Minimisation d'un AFD

Données : $B = (V, E, \{d\}, A, T)$, un AFD complet
Résultat : $M = (V, ME, \{nd\}, MA, MT)$, un AFD minimal
i=0;
Initialiser la partition $P_i = \{A, E - A\}$;
répéter
  pour chaque $C \in P_i$ **faire**
    si *il existe plusieurs états de C distinguables par un* $x \in V$ **alors**
      partitionner C en $C_1, C_2, \ldots, C_n$ dans P_{i+1} de manière à ce que ces sous-classes ne soient plus distinguables par x;
    sinon
      recopier C dans P_{i+1};
  i=i+1;
jusqu'à $P_i = P_{i-1}$;
numéroter chaque classe $C \in P_i$ pour former les états de ME;
le nouvel état de départ nd est le numéro de la classe qui contient d;
MA est l'ensemble des numéros de classes contenant des états d'arrivée de A;
MT est constitué des transitions entre les classes de P_i;
supprimer les états puits non finaux ainsi que les états non accessibles;

Remarquons qu'un état d'arrivée de M ne contient que des états d'arrivée de B à cause de la partition initiale.

Exemple 9
Soit un AFD complet :

$$B = (\{a, b\}, [1, 6], \{1\}, \{3, 4, 5\}, \{1a2, 1b3, 2a2, 2b3, 3a4, 3b6, 4a5, 4b6, 5a5, 5b6, 6a6, 6b6\})$$

On obtient la partition initiale : $P_0 = \{\{3, 4, 5\}, \{1, 2, 6\}\}$. *La classe* $\{3, 4, 5\}$ *n'est pas distinguable ni par a (classe* $\{3, 4, 5\}$*), ni par b (classe* $\{1, 2, 6\}$*). Par contre, la classe* $\{1, 2, 6\}$ *se distingue sur b. Par conséquent :*

$$P_1 = \{\{3, 4, 5\}, \{1, 2\}, \{6\}\} = P_2$$

Il ne reste plus qu'à supprimer la classe {6} qui est un puits non final pour obtenir l'AFD minimal :

$$M = (\{a, b\}, \{12, 345\}, \{12\}, \{345\}, \{12a12, 12b345, 345a345\})$$

Exercice 1 Soit l'expression régulière $(a|bc)*$. Calculer l'AFDM correspondant en passant par la construction de Thompson.

Chapitre 3

Analyse syntaxique

L'analyse syntaxique du programme source doit vérifier que celui-ci est bien un mot du langage de programmation. Pour cela, la grammaire du langage est utilisée. Cette grammaire $G = (V_T, V_N, R, S)$ est algébrique (insensible au contexte). Toutes les règles de R sont donc de la forme : $X \to \alpha$ avec $X \in V_N$ et $\alpha \in (V_T \cup V_N)^*$. De plus, G doit être non ambigüe afin d'éviter différentes sémantiques pour un même programme. Ainsi, il existe une unique dérivation gauche depuis l'axiome S de la grammaire et conduisant au programme. C'est-à-dire qu'il existe un unique arbre de dérivation dont la frontière soit le programme. Cette analyse peut se faire selon deux approches :

- l'analyse syntaxique descendante consiste à étudier l'unique dérivation gauche possible en partant de l'axiome et en allant vers le programme. L'arbre de dérivation est construit (ou pas) depuis la racine S vers les feuilles.
- L'analyse syntaxique ascendante consiste, au contraire, à partir du programme et à remonter vers l'axiome S. L'arbre de dérivation est construit (ou pas) depuis les feuilles vers la racine S.

De plus, la phase d'analyse syntaxique peut générer selon les cas :

- un résultat booléen indiquant la correction syntaxique. C'est le cas des vérificateurs syntaxiques tels que `lint`, qui est un vérificateur pour le C.
- Un arbre syntaxique représentant le programme. Celui-ci est soit un arbre de dérivation (arbre complet), soit un arbre abstrait (arbre simplifié). Cet arbre servira ensuite pour l'analyse sémantique puis la synthèse de la cible.
- Le programme cible directement compilé par la phase d'analyse syntaxique. On parle de traduction dirigé par la syntaxe. Cette traduction utilise fréquemment des grammaires attribuées.
- Le résultat de l'évaluation du programme source. C'est le cas des interpréteurs de programme et des évaluateurs d'expressions (calculette).

3.1 Analyse descendante récursive

C'est une méthode de programmation qui associe une fonction, pouvant être récursive, à chaque symbole non terminal de la grammaire. Ces fonctions s'appellent suite à la reconnaissance de certains jetons du flot d'entrée correspondant aux début des parties droites des règles de production. Ces jetons permettent donc de prédire la règle de production à choisir. Il est nécessaire que la grammaire possède un certains nombre de propriétés pour permettre l'analyse descendante prédictive.

La propriété fondamentale des grammaires pouvant donner lieu à l'analyse descendante est la **non récursivité à gauche**. En effet, celle-ci générerait des appels récursifs infinis. La récursivité à droite étant permise, il est toujours possible de transformer une grammaire récursive à gauche en une grammaire équivalente non récursive à gauche.

Le nombre de symboles terminaux nécessaires à la prédiction de la règle de production à choisir est une caractéristique des analyses descendantes prédictives. Dans le cas où ce nombre est 0, on choisit une production quelconque et on tente la descente. Si celle-ci échoue, il faudra revenir sur le(s) choix effectués (backtracking). Le backtracking étant coûteux du point de vue de l'efficacité, on utilise toujours au moins un symbole (jeton) de prédiction (prévision). Ce jeton doit être lu avant d'entrer dans une fonction afin de permettre le retour sans effet dans le cas d'une production epsilonesque.

Exemple 10
Soit la grammaire d'expressions arithmétiques $G_E = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E\}, R, E)$ *avec les règles de R suivantes :*

$$E \quad \to \quad E + E | E * E | (E) | 0 | 1 | \ldots | 9$$

Cette grammaire G_E *étant ambigüe, on écrit une grammaire équivalente non ambigüe selon le schéma Expression Terme Facteur (ou ETF) :*

- *une expression est quelconque, par exemple 1+2*3+4 ;*
- *un terme est un élément d'une somme : dans l'exemple précédent, 1, 2*3 et 4 sont trois termes ;*

— *un facteur est un élément d'un produit : dans l'exemple précédent, 2 et 3 sont des facteurs du produit 2*3.*

$G_{ETF} = (\{0,1,\ldots,9,+,*,(,)\},\{E,T,F\},R,E)$ *avec les règles de* R *suivantes :*

$$\begin{aligned} E &\rightarrow E+T|T \\ T &\rightarrow T*F|F \\ F &\rightarrow (E)|0|1|\ldots|9 \end{aligned}$$

Cette grammaire G_{ETF} *n'est pas ambiguë : pour un même niveau de parenthésage, les opérateurs* $+$ *doivent être tous générés avant de générer un opérateur *.* G_{ETF} *étant récursive à gauche, on écrit une grammaire équivalente non récursive à gauche* $G_{ENR} = (\{0,1,\ldots,9,+,*,(,)\},\{E,R,T,S,F\},X,E)$ *avec les règles de* X *suivantes :*

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow +TR|\varepsilon \\ T &\rightarrow FS \\ S &\rightarrow *FS|\varepsilon \\ F &\rightarrow (E)|0|1|\ldots|9 \end{aligned}$$

Enfin, il reste à écrire un vérificateur (reconnaisseur) syntaxique récursif utilisant un jeton de prédiction. Le programme C suivant effectue cette vérification syntaxique en calquant la structure de ses fonctions sur la grammaire G_{ENR}.

```
/** @file analdesc.c
 *@author Michel Meynard
 *@brief Analyse descendante récursive d'expression arithmétique
 *
 * Ce fichier contient un reconnaisseur d'expressions arithmétiques composée de
 * littéraux entiers sur un car, des opérateurs +, * et du parenthésage ().
 * Remarque : soit rediriger en entrée un fichier, soit terminer par deux
 * caractères EOF (Ctrl-D), un pour lancer la lecture, l'autre comme "vrai" EOF.
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
                            /* les macros sont des blocs : pas de ';' apres */
#define AVANCER {jeton=getchar();numcar++;}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu : erreur de syntaxe \
au caractère numéro %d \n",numcar); exit(1);}

void E(void);void R(void);void T(void);void S(void);void F(void); /* déclars */

int jeton;                       /* caractère courant du flot d'entrée */
int numcar=0;                    /* numero du caractère courant (jeton) */

void E(void){
  T();                          /* regle : E->TR */
  R();
}
void R(void){
  if (jeton=='+') {             /* regle : R->+TR */
    AVANCER
    T();
    R();
  }
  else ;                        /* regle : R->epsilon */
}
void T(void){
  F();
  S();                          /* regle : T->FS */
}
void S(void){
```

```
  if (jeton=='*') {             /* regle : S->*FS */
    AVANCER
    F();
    S();
  }
  else ;                        /* regle : S->epsilon */
}
void F(void){
  if (jeton=='(') {             /* regle : F->(E) */
    AVANCER
    E();
    TEST_AVANCE(')')
  }
  else
    if (isdigit(jeton))         /* regle : F->0|1|...|9 */
      AVANCER
    else ERREUR_SYNTAXE
}
int main(void){                 /* Fonction principale */
  AVANCER /* initialiser jeton sur le premier car */
  E();                          /* axiome */
  if (jeton==EOF)               /* expression reconnue et rien après */
    printf("\nMot reconnu\n");
  else ERREUR_SYNTAXE           /* expression reconnue mais il reste des car */
  return 0;
}
```

L'exécution de ce vérificateur donne les résultats suivants :

```
>analdesc
1+2*3+(4+(5*(2+(1)+2)*3))<Ctrl>-<D>
Mot reconnu
>analdesc
1+2*4)+5<Ctrl>-<D>
Mot non reconnu : erreur de syntaxe au caractère numéro 6
```

Exercice 2 Ecrire un vérificateur syntaxique pour le langage de Dyck à un couple de parenthèses : $S \rightarrow SS|aSb|\varepsilon$

3.2 Analyse descendante par automate à pile

3.2.1 Introduction

Un automate à pile est une machine lisant itérativement des symboles terminaux (jetons) depuis le flot d'entrée, gérant une pile de symboles, et exécutant des actions en fonction d'une table d'analyse ou table d'actions. Le flot d'entrée est constitué d'une suite de jetons terminée par un symbole spécial de fin symboliquement représenté par $ (jeton 0 retourné par yylex()). La pile est toujours initialisée avec le symbole spécial $ puis est manipulée par des empilements et dépilements dépendant de la table d'actions. La table d'actions est une table à 2 dimensions indicées par les non terminaux d'une part, et les symboles terminaux (jetons du flot) et $ d'autre part. Ainsi, en fonction du symbole de sommet de pile et du jeton courant, la table indique l'action à réaliser.

Les automates à pile sont utilisés en analyse descendante comme en ascendante avec des différences au niveau des types d'actions et des types de symboles de pile. En analyse descendante, la pile de l'automate simule les appels récursifs des fonctions.

3.2.2 Fonctionnement de l'automate à pile en analyse descendante

La table $M[V_N, V_T \cup \{\$\}]$ contient une règle de production ou l'action ERREUR dans chacune de ces cases. A tout moment, l'analyse du flot d'entrée consiste à regarder la règle de production correspondant au sommet de pile et au jeton d'entrée. Puis, selon les cas, l'automate soit :

— s'arrête en générant une erreur de syntaxe,
— avance sur le flot et dépile un jeton,
— empile **à l'envers** la partie droite de la règle,
— termine en indiquant la réussite de l'analyse.

Algorithme 7 : Fonctionnement de l'automate

Données : Une table d'analyse $M[V_N, V_T \cup \{\$\}]$, un flot de jetons terminé par \$, une grammaire $G = (V_T, V_N, R, S)$
Résultat : Erreur ou Succès

```
Pile=construirePileVide() // contenu : terminaux, non terminaux et $
empiler(Pile,$) // initialisation
empiler(Pile,S) // l'axiome de la grammaire
jeton=lireFlot() // jeton courant du flot
tant que vrai faire
    si sommet(Pile)=jeton et jeton=$ alors
        terminer l'algorithme avec succès // return true
    sinon
        si sommet(Pile)=jeton alors
            dépiler(Pile) // avançons
            jeton=lireFlot() // jeton suivant du flot
        sinon
            si sommet(Pile)∈ V_T ∪ {$} alors
                terminer l'algorithme en échec // return false
            sinon
                si M[sommet(Pile), jeton] = ERREUR alors
                    terminer l'algorithme en échec // return false
                sinon
                    s=sommet(Pile)
                    dépiler(Pile) // remplaçons le non terminal
                    empiler dans Pile la partie droite de la règle en M[s, jeton] de droite à gauche
```

L'algorithme 7 précise le fonctionnement exact de l'automate à pile.

Exemple 11
Un exemple simple de fonctionnement d'une analyse descendante à l'aide d'un automate à pile consiste à étudier une grammaire de Dyck à un couple de parenthèses. Soit la grammaire $G_D = (\{a, b\}, \{S\}, R, S)$ avec les règles de R suivantes :

$$S \to aSbS|\varepsilon$$

On obtient la table d'analyse suivante (voir algorithme 15) :

	a	b	\$
S	$S \to aSbS$	$S \to \varepsilon$	$S \to \varepsilon$

Etudions le fonctionnement de l'automate, c'est-à-dire de sa pile, sur le mot d'entrée abaababb\$ indicé à partir de 1 :

indice	*1a*	*1a*	*2b*	*2b*	*3a*	*3a*	*4a*	*4a*	*5b*	*5b*	*6a*	*6a*	*7b*	*7b*	*8b*	*8b*	*9\$*	*9\$*
								a				a						
								S	S			S	S					
		a				a		b	b	b		b	b	b				
		S	S			S	S	S	S	S	S	S	S	S	S			
		b	b	b		b	b	b	b	b	b	b	b	b	b	b		
	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	
	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$

Remarquons encore une fois que les empilements de partie droite de règle se font à l'envers, c'est-à-dire de droite à gauche.

3.2.3 Algorithmique

La grammaire doit posséder certaines propriétés de forme de ses règles afin de permettre l'analyse descendante. Nous allons examiner les différentes transformations de règles susceptibles de mettre une grammaire $G = (V_T, V_N, R, S)$ quelconque en "bonne forme", c'est-à-dire non récursive à gauche, non ambiguë et factorisée ! Attention, la désambiguation d'une grammaire étant non décidable, celle-ci devra être réalisée par une méthode ad hoc.

Suppression des ε-productions

Les symboles non terminaux **effaçables**, c'est-à-dire pouvant dériver en ε, sont détectés de la manière suivante. Un symbole non terminal effaçable :
— soit dérive directement en ε,
— soit dérive en un mot constitué exclusivement de symboles non terminaux effaçables.
Soit $G = (V_T, V_N, R, S)$, soit E_i une suite d'ensembles Effaçables de symboles non terminaux définie comme suit :
— $E_1 = \{X \in V_N / (X \to \varepsilon) \in R\}$
— $E_{i+1} = E_i \cup \{X \in V_N / (X \to \alpha) \in R \text{ et } \alpha \in {E_i}^*\}$
On prouve que les ensembles E_i ne contiennent que des symboles non terminaux effaçables, c'est à dire dérivant en ε. On prouve également que la suite E_i converge et est donc constante au-delà d'un certain rang n : $\exists n \in \mathbb{N}, E_n = E_{n+k}, \forall k \in \mathbb{N}$. Par conséquent, $\forall X \in V_N, X \overset{*}{\Rightarrow} \varepsilon$ si et seulement si $X \in E_n$.

Il reste à construire une grammaire G_{SE} ne contenant (presque) plus d'ε-production et équivalente à G. Il peut rester une ε-production dans le cas où le langage de la grammaire contient le mot vide...

Soit $G_{0E} = (V_T, V_N, R_1, S)$ avec un ensemble de règles défini comme suit :

$R_1 = \{X \to \alpha$ tel que $\alpha \neq \varepsilon$ et $\exists X \to \beta \in R$ tel que α s'obtient à partir de β en supprimant un nombre quelconque $(k \in [0, |\beta|[)$ d'occurrences d'éléments effaçables (de $E_n)\}$

On prouve que $L(G_{0E}) = L(G) - \{\varepsilon\}$. Si S est un symbole effaçable de G, $S \in E_n$, on obtient G_{SE} en ajoutant un nouvel axiome S_1 et deux nouvelles règles :

$G_{SE} = (V_T, V_N \cup \{S_1\}, R_1 \cup \{S_1 \to \varepsilon | S\}, S_1)$

Sinon, $S \notin E_n$, on a $G_{SE} = G_{0E}$.

Exemple 12
Soit la grammaire $G = (\{a, b\}, \{S, X, Y\}, R, S)$ avec les règles de R suivantes :

$$\begin{aligned} S &\to aX|Y|XX \\ X &\to \varepsilon|b|XX \\ Y &\to aXb \end{aligned}$$

On calcule les ensembles d'effaçables : $E_1 = \{X\}, E_2 = \{X, S\}, E_3 = \{X, S\}$. On obtient donc un nouvel ensemble de règles R_1 :

$$\begin{aligned} S &\to aX|a|Y|XX|X \\ X &\to b|XX|X \\ Y &\to aXb|ab \end{aligned}$$

Pour finir, voici la grammaire équivalente à G et ne contenant qu'une ε-production :

$G_{SE} = (V_T, V_N \cup \{S_1\}, R_1 \cup \{S_1 \to \varepsilon | S\}, S_1)$.

Remarquons que notre construction n'admet au plus qu'une ε-production et que celle-ci se trouve en partie droite de l'axiome qui ne peut lui-même être atteint par aucune autre production.

Dans les algorithmes suivants on supposera l'inexistence d'ε-production et/ou de cycle ($X \overset{+}{\Rightarrow} X$). Remarquons d'abord qu'il ne peut exister de cycle sur X_1. Si la grammaire G_{SE} possède, $S_1 \to \varepsilon|S$, on appliquera ces algorithmes à la grammaire $G_{0E} = (V_T, V_N, R_1, S)$, puis on rajoutera l'axiome S_1 et ses deux règles tout à fait à la fin du processus.

Suppression des cycles

On suppose une grammaire sans ε-production. L'algorithme 8 supprime les cycles de dérivation : $X \overset{+}{\Rightarrow} X$. Une production est appelée **substitution de non terminal** ou plus simplement substitution lorsqu'elle est de la forme : $X \to Y$. Seules les substitutions engendrant des cycles doivent être supprimées. Dans l'algorithme 8, on calcule la Fermeture Transitive des non terminaux Substituables à chaque symbole non terminal. Ce calcul partitionne V_N en classes d'équivalence correspondant aux cycles de non terminaux substituables. Puis on filtre les productions selon l'appartenance de leur partie gauche à un cycle.

La preuve de l'élimination des cycles effectuée par cet algorithme tient à ce que les seules règles de substitutions $(X_i \to X_j)$ autorisées dans R_{SC} impliquent que X_i et X_j ne soient pas dans le même cycle. Remarquons que les non terminaux membres d'un même cycle peuvent être représentés par un seul non terminal du cycle car ils auront tous les mêmes règles de production.

Exemple 13
Soit la grammaire $G = (\{a, b, c, d\}, \{X_1, X_2, X_3\}, R, X_1)$ avec les règles de R suivantes :

$$\begin{aligned} X_1 &\to X_2|a \\ X_2 &\to X_1|X_2|X_3|b \\ X_3 &\to bX_1|X_2a \end{aligned}$$

Algorithme 8 : Suppression des cycles

Données : $G_{0E} = (V_T, V_N = \{X_1, X_2, \ldots, X_n\}, R, S)$ une grammaire sans ε-production
Résultat : $G_{SC} = (V_T, V_N, R_{SC}, S)$ une grammaire sans cycle
$R_{SC} = \emptyset$ /* *initialisation* */
Construire la Fermeture Transitive des non terminaux Substituables à chaque $X_i \in V_N$:
$FTS(X_i) = \{X_j \in V_N / X_i \stackrel{+}{\Rightarrow} X_j\}$
pour *i=1 à n* **faire**
  si $X_i \notin FTS(X_i)$ /* pas de cycle */ **alors**
    pour chaque *production* $X_i \to \alpha \in R$ **faire**
      $R_{SC} = R_{SC} \cup \{X_i \to \alpha\}$ /* *ne rien faire* */
  sinon
    pour chaque $X_j \in FTS(X_i)$ /* traitons les non terminaux substituables, y compris X_i */ **faire**
      si $X_i \notin FTS(X_j)$ /* X_j pas dans le cycle */ **alors**
        $R_{SC} = R_{SC} \cup \{X_i \to X_j\}$
      sinon
        pour chaque *production* $X_j \to \alpha \in R$ **faire**
          si $|\alpha| > 1$ *ou* $\alpha[1] \in V_T$ **alors**
            $R_{SC} = R_{SC} \cup \{X_i \to \alpha\}$ /* *transitivité pour les non substitutions* */

On calcule les fermetures transitives des substituables : $FTS(X_1) = \{X_1, X_2, X_3\}$, $FTS(X_2) = \{X_1, X_2, X_3\}$, $FTS(X_3) = \emptyset$. *On obtient donc un nouvel ensemble de règles sans cycle* R_{SC} *:*

$$\begin{aligned} X_1 &\to a|b|X_3 \\ X_2 &\to a|b|X_3 \\ X_3 &\to bX_1|X_2a \end{aligned}$$

Remarquons que X_1 *et* X_2 *peuvent être remplacés par* X_1 *qui les représente tous deux. Ce qui donne :*

$$\begin{aligned} X_1 &\to a|b|X_3 \\ X_3 &\to bX_1|X_1a \end{aligned}$$

Suppression de la récursivité à gauche immédiate

Une récursivité à gauche immédiate d'un symbole non terminal X se matérialise par au moins une règle de production $X \to X\alpha$. La suppression de cette récursivité à gauche immédiate nécessite de transformer l'ensemble des règles de production ayant X comme partie gauche (les X-productions). L'algorithme 9 réalise cette transformation.

Remarquons que l'appel de cet algorithme nécessite d'avoir au moins une récursivité à gauche immédiate ($n \neq 0$) et au moins une autre production ($k \neq 0$). Cette dernière condition est indispensable dans une grammaire sans ε-production. Sinon, le non terminal X ne peut dériver en un mot terminal !

Algorithme 9 : Suppression de la récursivité à gauche immédiate

Données : Un ensemble de productions : $P = X \to X\alpha_1|X\alpha_2|\ldots|X\alpha_n|\beta_1|\beta_2|\ldots|\beta_k$ sans ε-production et telles que $n \neq 0$ et $k \neq 0$
Résultat : Un nouveau symbole non terminal R_X et un ensemble de productions P' sans récursivité à gauche immédiate
$P' = \{R_X \to \varepsilon\}$ // *initialisation*
pour *i=1 à k* **faire**
  $P' = P' \cup \{X \to \beta_i R_X\}$
pour *j=1 à n* **faire**
  $P' = P' \cup \{R_X \to \alpha_j R_X\}$

L'algorithme 9 crée un nouveau symbole R_X (Reste de X), pour remplacer la récursivité à gauche par une récursivité à droite sur R_X. Remarquons que R_X possède une ε-production donc est effaçable. La correction de l'algorithme, c'est-à-dire l'équivalence des deux ensembles de productions P et P', se démontre par une double récurrence sur i et j.

Exemple 14
Soit la grammaire d'expressions arithmétiques $G_E = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E\}, P, E)$ *avec les règles de P suivantes :*

$$E \to E + E|E * E|(E)|0|1|\ldots|9$$

Après application de l'algorithme 9, on obtient la grammaire suivante : $G_{ENRI} = (\{0,1,\ldots,9,+,*,(,)\},\{E,R_E\},P',E)$ *avec les règles de* P' *suivantes :*

$$\begin{array}{rcl} E & \rightarrow & (E)R_E|0R_E|1R_E|\ldots|9R_E \\ R_E & \rightarrow & \varepsilon|+ER_E|*ER_E \end{array}$$

Remarquons que G_{ENRI} *n'est plus récursive à gauche, mais elle reste ambiguë.*

Exercice 3 Soit la grammaire $G_{ETF} = (\{0,1,\ldots,9,+,*,(,)\},\{E,T,F\},R,E)$ avec les règles de R suivantes :

$$\begin{array}{rcl} E & \rightarrow & E+T|T \\ T & \rightarrow & T*F|F \\ F & \rightarrow & (E)|0|1|\ldots|9 \end{array}$$

Supprimer la récursivité à gauche dans cette grammaire.

Dans certains cas, la suppression de la récursivité à gauche immédiate ne suffit pas car il peut subsister des récursivités plus complexes : dans les productions $X_1 \rightarrow X_2a|a, X_2 \rightarrow X_1b|b$ il n'y pas de récursivité à gauche immédiate mais il y a de la récursivité à gauche !

Suppression de la récursivité à gauche

L'algorithme 10 s'applique à une grammaire sans cycle, sans ε-production et sans récursivité à gauche immédiate. Il produit une grammaire sans récursivité à gauche, c'est-à-dire sans dérivation de la forme $X \stackrel{+}{\Rightarrow} X\alpha$.

Algorithme 10 : Suppression de la récursivité à gauche

Données : $G = (V_T, V_N = \{X_1, X_2, \ldots, X_n\}, R, S)$ une grammaire sans cycle, sans ε-production et sans récursivité à gauche immédiate
Résultat : $G_{NR} = (V_T, V_{NR}, R_{NR}, S)$ une grammaire sans récursivité à gauche
$R_{NR} = \emptyset$
pour *i=1 à n* **faire**
  $P = \{X_i \rightarrow \gamma \in R\}$ // *ensemble des productions* $X_i \rightarrow \ldots$
  tant que $\exists X_i \rightarrow X_j\alpha \in P$ *telle que* $i > j$ **faire**
    $P = P - \{X_i \rightarrow X_j\alpha\}$ // *suppression*
    pour chaque *production* $X_j \rightarrow \beta \in R_{NR}$ **faire**
      $P = P \cup \{X_i \rightarrow \beta\alpha\}$ // *remplacement*
  P'=Supprimer la récursivité immédiate dans P (algo. 9)
  $R_{NR} = R_{NR} \cup P'$

La preuve de la correction de l'algorithme tient en ce qu'à la fin, il est impossible d'avoir une production de la forme $X_i \rightarrow X_j\alpha$ telle que $i \geq j$.

Remarquons qu'il est toujours possible mais pas toujours nécessaire, en analyse descendante, de transformer la grammaire initiale de la façon suivante :

1. suppression des ε-productions,
2. suppression des cycles,
3. suppression des récursivités à gauche immédiates,
4. suppression des récursivités à gauche.

La seule propriété à respecter est la non récursivité à gauche. Le moyen par lequel on obtient cette propriété est indifférent. Remarquons qu'après la dérécursivation, on obtient souvent des grammaires ayant des ε-productions. Ainsi, dans l'exemple 10, la grammaire G_{ENR} est non récursive à gauche et contient des ε-productions. Ceci n'est pas génant. En effet, ces productions ne peuvent en aucun cas impliquer une récursivité à gauche d'un non terminal.

Exemple 15
Soit la grammaire $G = (\{a,b,d\},\{X_1,X_2,X_3\},P,X_1)$ *avec les règles de* P *suivantes :*

$$\begin{array}{rcl} X_1 & \rightarrow & X_2a|d \\ X_2 & \rightarrow & X_3a|X_1b \\ X_3 & \rightarrow & X_1a \end{array}$$

Après application de l'algorithme 10, on obtient la grammaire suivante $G' = (\{a, b, d\}, \{X_1, X_2, R_2, X_3, R_3\}, P', X_1)$ *avec les règles de* P' *suivantes :*

$$\begin{aligned} X_1 &\rightarrow X_2a|d \\ X_2 &\rightarrow X_3aR_2|dbR_2 \\ R_2 &\rightarrow \varepsilon|abR_2 \\ X_3 &\rightarrow dbR_2aaR_3|daR_3 \\ R_3 &\rightarrow \varepsilon|aR_2aaR_3 \end{aligned}$$

Factorisation à gauche

Si plusieurs parties droites de X-productions ont même préfixe, la prédiction de la règle à choisir est retardée jusqu'à ce qu'un jeton permette de déterminer la "bonne" règle. Il faudra donc pouvoir lire plusieurs jetons en avance. La factorisation des parties droites est destinée à réduire à 1 ce nombre de jetons de prévision.

Algorithme 11 : Factorisation à gauche

Données : $G = (V_T, V_N = \{X_1, X_2, \ldots, X_n\}, R, S)$ une grammaire
Résultat : $G_F = (V_T, V_F, R_F, S)$ une grammaire factorisée à gauche
$V_F = V_N$ *// initialisation*
$R_F = R$
pour chaque *symbole non terminal X non marqué de* V_F **faire**
  calculer α, le plus long préfixe commun des parties droites des X-productions de R_F
  tant que $\alpha \neq \varepsilon$ **faire**
    $V_F = V_F \cup \{X'\}$ *// nouveau non terminal*
    soit $X \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma_1|\ldots|\gamma_k$ l'ensemble des X-productions de R_F
    remplacer ces productions par : $\{X \rightarrow \alpha X'|\gamma_1|\ldots|\gamma_k, X' \rightarrow \beta_1|\beta_2|\ldots|\beta_n\}$
    calculer α, le plus long préfixe commun des parties droites des X-productions de R_F
  marquer X

Exemple 16
Soit la grammaire du "if then else" $G = (\{i, t, e, a, b\}, \{S, E\}, R, S)$ *avec les règles de* R *suivantes :*

$$\begin{aligned} S &\rightarrow iEtS|iEtSeS|a \\ E &\rightarrow b \end{aligned}$$

Après application de l'algorithme 11, on obtient la grammaire : $G_F = (\{i, t, e, a, b\}, \{S, S', E\}, R_F, S)$ *avec les règles de* R_F *suivantes :*

$$\begin{aligned} S &\rightarrow iEtSS'|a \\ S' &\rightarrow \varepsilon|eS \\ E &\rightarrow b \end{aligned}$$

Remarquons que cette grammaire factorisée reste ambiguë, ce qui posera problème à l'analyse.

Premiers

La fonction `premiers` est nécessaire à la construction de la table d'analyse qu'utilise l'automate à pile. Elle retourne un ensemble de terminaux (jetons). `premiers` suppose une grammaire non récursive à gauche mais pouvant admettre des ε-productions.

La fonction `premiers(`α`)` retourne l'ensemble des terminaux qui débute un mot dérivant de α. Si α est effaçable alors ε fait partie de ses `premiers`. Pour calculer `premiers(`α`)`, il faut commencer par calculer `premiers(`X`)`, quel que soit X un symbole de V. L'algorithme 12 réalise cette fonction.

L'algorithme 12 est trivial pour les terminaux. Pour les non terminaux, il consiste à accumuler les `premiers(`Y_i`)` tant que Y_{i-1} est effaçable. ε n'est ajouté que dans le cas ou une partie droite de production est entièrement effaçable.

Exemple 17
Soit la grammaire non récursive à gauche $G_{ENR} = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E, R, T, S, F\}, X, E)$ *avec les règles de* X *suivantes :*

$$E \rightarrow TR$$

Algorithme 12 : premiers(X)

Données : $X \in V$ un symbole de $V_T \cup V_N$, et une grammaire non récursive à gauche $G = (V_T, V_N, R, S)$
Résultat : $Resultat \subseteq V_T \cup \{\varepsilon\}$ un ensemble de terminaux
si $X \in V_T$ **alors**
  retourner $\{X\}$
sinon
  $Resultat = \emptyset$ // *initialisation*
  pour chaque *production* $X \to Y_1 Y_2 \dots Y_k \alpha$ *telle que* $Y_i \in V_N$ *et* $\alpha \in \{\varepsilon\} \cup V_T \bullet V^*$ **faire**
    si $k = 0$ *et* $\alpha = \varepsilon$ **alors**
      $Resultat = Resultat \cup \{\varepsilon\}$ // *ε-production*
    sinon
      si $k = 0$ **alors**
        $Resultat = Resultat \cup \{\alpha[1]\}$
      sinon
        $Resultat = Resultat \cup (\texttt{premiers}(Y_1) - \{\varepsilon\})$ // *non réc. gauche*
        i=1
        tant que $i \leq k$ *et* Y_i *est effaçable* **faire**
          $i = i + 1$
          $Resultat = Resultat \cup (\texttt{premiers}(Y_i) - \{\varepsilon\})$ // *non réc. gauche*
        si $i = k + 1$ **alors**
          si $|\alpha| = 0$ **alors**
            $Resultat = Resultat \cup \{\varepsilon\}$ // *tous les* Y_i *s'effacent*
          sinon
            $Resultat = Resultat \cup \{\alpha[1]\}$
  retourner Resultat

$$
\begin{aligned}
R &\to +TR|\varepsilon \\
T &\to FS \\
S &\to *FS|\varepsilon \\
F &\to (E)|0|1|\dots|9
\end{aligned}
$$

On obtient par l'application de l'algorithme 12 :

$$
\begin{aligned}
\texttt{premiers}(F) &= \{(, 0, 1, \dots, 9\} \\
\texttt{premiers}(S) &= \{*, \varepsilon\} \\
\texttt{premiers}(T) &= \texttt{premiers}(F) \\
\texttt{premiers}(R) &= \{+, \varepsilon\} \\
\texttt{premiers}(E) &= \texttt{premiers}(F)
\end{aligned}
$$

Remarquons la récursivité de l'algorithme et la terminaison de celui-ci uniquement si la grammaire est non récursive à gauche. Cette propriété reste fondamentale pour le calcul des `premiers(`α`)` qui fait appel aux `premiers(`X`)`. L'algorithme 13 calcule justement ces `premiers(`α`)`.

Suivants

L'algorithme 14 est nécessaire à la construction de la table d'analyse qu'utilise l'automate à pile. Il utilise une grammaire G et calcule un tableau d'ensembles de terminaux, et éventuellement \$ le symbole de fin d'entrée. Chaque case du tableau est associé à un non terminal de G. Son contenu est l'ensemble des terminaux pouvant suivre immédiatement ce symbole non terminal X_i de G dans un mot dérivant de l'axiome : $TabSuivants[X_i] = \{x \in V_T \cup \{\$\} / S \overset{*}{\Rightarrow} \alpha X_i x \beta\}$. L'algorithme 14 calcule ce tableau $TabSuivants[X_i]$.

Exemple 18
Soit la grammaire non récursive à gauche G_{ENR} de l'exemple 17. On obtient par l'application de l'algorithme 14 :

$$
\begin{aligned}
TabSuivants[E] &= \{\$,)\} \\
TabSuivants[T] &= \{+, \$,)\} \\
TabSuivants[R] &= \{\$,)\} \\
TabSuivants[F] &= \{*, +, \$,)\} \\
TabSuivants[S] &= \{+, \$,)\}
\end{aligned}
$$

Algorithme 13 : premiers(α)

Données : $\alpha = Y_1 Y_2 \dots Y_k$ avec $Y_i \in V$, ainsi qu'une grammaire non récursive à gauche $G = (V_T, V_N, R, S)$
Résultat : *Resultat* $\subseteq V_T \cup \{\varepsilon\}$ un ensemble de terminaux
si $\alpha = \varepsilon$ **alors**
  retourner $\{\varepsilon\}$
sinon
  Resultat $= \emptyset$ *// initialisation*
  Resultat = *Resultat* $\cup$ (`premiers(`Y_1`)` $- \{\varepsilon\}$)
  i=1
  tant que $i \leq k$ *et* $\varepsilon \in$ `premiers(`Y_i`)` **faire**
    $i = i + 1$
    Resultat = *Resultat* $\cup$ (`premiers(`Y_i`)` $-\{\varepsilon\}$) *// non réc. gauche*
  si $i = k + 1$ **alors**
    Resultat = *Resultat* $\cup \{\varepsilon\}$ *// tous les* Y_i *s'effacent*
  retourner Resultat

Algorithme 14 : Suivants

Données : $G = (V_T, V_N = \{X_1, X_2, \dots, X_n\}, R, X_1)$, une grammaire
Résultat : un tableau $TabSuivants[X_i]$ d'ensembles de terminaux $\{x_1, x_2, \dots, x_m\} \subseteq (V_T \cup \{\$\})$
$TabSuivants[X_1] = \{\$\}$ *// initialisation pour l'axiome*
pour *i=2 à n* **faire**
  $TabSuivants[X_i] = \emptyset$ *// initialisation*
répéter
  stable=vrai *// booléen testant la stabilité du tableau*
  pour chaque *production* $Y \to \gamma$ *de* R **faire**
    pour chaque *non terminal* X *de* $\gamma : Y \to \alpha X \beta$ *avec* $\gamma = \alpha X \beta$ **faire**
      si $\beta = \varepsilon$ **alors**
        si $TabSuivants[Y] \not\subseteq TabSuivants[X]$ **alors**
          stable=faux
          $TabSuivants[X] = TabSuivants[X] \cup TabSuivants[Y]$
      sinon
        si `premiers(`β`)` $- \{\varepsilon\} \not\subseteq TabSuivants[X]$ **alors**
          stable=faux
          $TabSuivants[X] = TabSuivants[X] \cup$ (`premiers(`β`)` $- \{\varepsilon\}$)
        si $\varepsilon \in$ `premiers(`β`)` // β est effaçable **alors**
          si $TabSuivants[Y] \not\subseteq TabSuivants[X]$ **alors**
            stable=faux
            $TabSuivants[X] = TabSuivants[X] \cup TabSuivants[Y]$
jusqu'à *stable*;

3.2.4 Construction de la table d'analyse

L'algorithme 15 réalise la construction de la table d'analyse qu'utilise l'automate à pile. Dans cette table, l'existence de plus d'une production dans une case est appelée un **conflit** et signifie que l'automate à pile aura un choix à réaliser ! Ceci n'est pas envisageable pour des raisons d'efficacité (backtrack).

Algorithme 15 : Construction de la table d'analyse

Données : Une grammaire $G = (V_T, V_N, R, S)$
Résultat : Une table d'analyse $M[V_N, V_T \cup \{\$\}]$ contenant des ensembles de productions
pour chaque *case* $M[i,j]$ **faire**
  $M[i,j] = \emptyset$
pour chaque *production* $X \to \alpha$ **faire**
  pour chaque $x \in$ `premiers(`α`)` $- \{\varepsilon\}$ **faire**
    $M[X,x] = M[X,x] \cup X \to \alpha$
  si $\varepsilon \in$ `premiers(`α`)` **alors**
    pour chaque $y \in TabSuivants[X]$ **faire**
      $M[X,y] = M[X,y] \cup X \to \alpha$
pour chaque *case* $M[i,j] == \emptyset$ **faire**
  $M[i,j] = \{ERREUR\}$

Exemple 19
Reprenons l'exemple 11 de la grammaire de Dyck à un couple de parenthèses. Soit la grammaire $G_D = (\{a,b\}, \{S\}, R = \{S \to aSbS|\varepsilon\}, S)$. *La première règle* $S \to aSbS$ *ne pose aucun problème car* `premiers(`$aSbS$`)` $= a$, *donc* $M[S,a] = S \to aSbS$. *Quant à la seconde production* $S \to \varepsilon$, *elle génère le calcul de* $TabSuivants[S] = \{b, \$\}$. *On obtient donc la table d'analyse suivante :*

	a	b	$\$$
S	$S \to aSbS$	$S \to \varepsilon$	$S \to \varepsilon$

Reprenons la grammaire plus complexe de l'exemple 17 et voyons la table d'analyse générée.

Exemple 20
Soit la grammaire non récursive à gauche $G_{ENR} = (\{0,1,\ldots,9,+,*,(,)\}, \{E,R,T,S,F\}, X, E)$ *avec les règles de* X *suivantes :*

$$\begin{aligned} E &\to TR \\ R &\to +TR|\varepsilon \\ T &\to FS \\ S &\to *FS|\varepsilon \\ F &\to (E)|0|1|\ldots|9 \end{aligned}$$

Il nous faut rappeler les `premiers()` *des non terminaux débutant des parties droites :*

$$\begin{aligned} \texttt{premiers(}F\texttt{)} &= \{(,0,1,\ldots,9\} \\ \texttt{premiers(}T\texttt{)} &= \texttt{premiers(}F\texttt{)} \end{aligned}$$

Il nous faut également rappeler les suivants des non terminaux effaçables :

$$\begin{aligned} TabSuivants[R] &= \{\$,)\} \\ TabSuivants[S] &= \{+,\$,)\} \end{aligned}$$

On obtient finalement par l'application de l'algorithme 15, la table suivante :

	$0\|1\|\ldots\|9$	$($	$)$	$+$	$*$	$\$$
E	$E \to TR$	$E \to TR$				
R			$R \to \varepsilon$	$R \to +TR$		$R \to \varepsilon$
T	$T \to FS$	$T \to FS$				
S			$S \to \varepsilon$	$S \to \varepsilon$	$S \to *FS$	$S \to \varepsilon$
F	$F \to 0\|1\|\ldots\|9$	$F \to (E)$				

Si on choisit de placer un ensemble de productions et pas seulement une production, dans l'algorithme 15, c'est pour permettre à l'utilisateur de désambiguer l'analyse de certaines grammaires ambiguës en choisissant la règle à appliquer parmi celles qui sont proposées. L'exemple suivant illustre ce problème.

Exemple 21
Soit la grammaire du "if then else" après factorisation : $G_{IF} = (\{i, t, e, a, b\}, \{S, S', E\}, R_{IF}, S)$ *avec les règles de* R_{IF} *suivantes :*

$$\begin{array}{rcl} S & \rightarrow & iEtSS'|a \\ S' & \rightarrow & \varepsilon|eS \\ E & \rightarrow & b \end{array}$$

Après calcul, on obtient les `premiers()` *:*

$$\begin{array}{rcl} \texttt{premiers}(S) & = & \{i, a\} \\ \texttt{premiers}(S') & = & \{e, \varepsilon\} \\ \texttt{premiers}(E) & = & \{b\} \end{array}$$

Il nous faut également rappeler les suivants des non terminaux effaçables :

$$\begin{array}{rcl} TabSuivants[S] & = & \{e, \$\} \\ TabSuivants[S'] & = & \{e, \$\} \\ TabSuivants[E] & = & \{t\} \end{array}$$

On obtient finalement par l'application de l'algorithme 15, la table suivante :

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS, S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

Dans cette table, l'entrée $M[S', e]$ *contient deux productions possibles. Il faut, dans ce cas, choisir de conserver la production* $S' \rightarrow eS$ *pour deux raisons. D'abord, parce qu'en l'absence de cette production, la partie "else" ne serait jamais reconnu ! Ensuite, parce que que l'ambiguïté de la grammaire (à quel "if" associer le "else") est ainsi supprimée dans l'analyseur. En effet, la partie "else" sera toujours associée syntaxiquement au "if" le plus proche, ce qui correspond à la sémantique choisie par tous les langages de programmation.*

3.2.5 Grammaires LL(1)

Définition 3 *Une grammaire dont la table d'analyse peut être calculée et dont toutes les entrées ont une unique production ou bien ERREUR, est appelée LL(1).*

La signification de cet acronyme est :
— Left to Right scanning of the input,
— Leftmost derivation,
— 1 symbole de prévision.

Théorème 11 *Aucune grammaire ambiguë et aucune grammaire récursive à gauche n'est LL(1).*

Théorème 12 *Une grammaire G est LL(1) si et seulement si les conditions suivantes sont respectées. Quelle que soit* $X \rightarrow \alpha|\beta$*, deux productions de G :*
— *il n'existe pas deux dérivations de* α *et* β *ayant un préfixe commun terminal ;*
— *une partie droite seulement,* α *ou bien* β*, peut s'effacer ;*
— *si* α *peut s'effacer, alors* β *ne dérive pas en un mot ayant un préfixe commun terminal avec suivants(X).*

3.2.6 Conclusion sur l'analyse descendante

Examinons les grammaires qui ne sont pas LL(1). Toutes les grammaires ambiguës ne sont pas LL(1). Certaines grammaires non ambiguës ne sont pas LL(1). Par exemple, $G_2 = (\{a, b\}, \{S, A\}, \{S \rightarrow Ab|aa, A \rightarrow a\}, S)$ est une grammaire simple produisant 2 mots aa et ab et n'est pas LL(1). En effet, sur la lecture du premier a, on ne peut pas déterminer quelle production de S utiliser.

Cependant, on peut parfois utiliser un automate à pile en analyse descendante pour reconnaitre le langage généré par une grammaire non LL(1). Par exemple, la grammaire ambiguë du `si ... alors ... sinon ...` de l'exemple

21 génère une table d'analyse ayant un conflit. On peut *déterminiser* cette table en réussissant à reconnaître le même langage. Malheureusement, ce problème du choix est indécidable et nécessite donc une réflexion ad hoc. Dans l'exemple précédent de G_2, le choix de l'une ou de l'autre des productions de S à privilégier aboutit à un langage reconnu réduit de moitié !

D'un point de vue plus pratique, le problème principal des grammaires LL(1) résulte dans le fait qu'elles sont souvent obtenues par de multiples transformations qui les rendent difficilement lisibles pour le concepteur du langage. Aussi, les actions sémantiques qu'il faut associer à ces règles deviennent difficiles à mettre en oeuvre.

3.3 Un langage et un outil pour l'analyse syntaxique : yacc

Yacc ("Yet Another Compiler Compiler") est un outil d'analyse syntaxique permettant d'écrire des grammaires algébriques LALR(1) assez générales ("Look Ahead Left to right scanning of the input, Rightmost derivation in reverse, 1 look-ahead token"). Il génère un analyseur syntaxique ascendant utilisant un automate à pile. Associés à chaque règle de grammaire, des actions peuvent être associées. Ces actions sont des instructions d'un langage de programmation (C ou C++) ainsi que des actions spécifiques de yacc. Il existe de nombreuses versions de yacc, dont bison que nous utiliserons et qui est une version gratuite du projet GNU accessible sur le Web. Bien entendu, yacc peut être utilsé conjointement à lex qui fournit lui les jetons consommés par l'analyseur généré par yacc.

3.3.1 Un exemple

Soit la grammaire ambiguë d'expressions booléennes $G_B == (\{0, 1, \&, |, !, (,)\}, \{E\}, R, E)$ avec les règles de R suivantes :

$$E \rightarrow (E)|E'|'E|E\&E|!E|0|1$$

On va construire un vérificateur syntaxique, en utilisant yacc, reconnaissant les mots du langage généré par cette grammaire.

Exemple 22
Voici le source yacc obtenu :

```
%{ /* veriflog.y  */
  #include  <stdio.h>
  int yylex(void);  void yyerror(char *s);
%}
%%
expr    :       '(' expr ')'
        {}
        |       expr '|' expr
        {}
        |       expr '&' expr
        {}
        |       '!' expr
        {}
        | '0'
        {}
        |       '1'
        {}
        ;
%% /* debut des fonctions C */
int yylex(void) {       /* analyseur lexical filtrant les blancs */
  int c;
  while(((c=getchar())==' ') || (c=='\t'))
    ;
  return (c);
}
void yyerror(char *s) {  /* appelée par yyparse sur erreur de syntaxe */
  fprintf(stderr,"%s\n",s);
}
int main(void){             /* fonction principale */
  if (!yyparse())   /* appel à l'analyseur généré par yacc */
    printf("\nExpression reconnue\n");
```

```
  else
    printf("\nExpression non reconnue\n");
  return 0;
}
```

Après compilation bison, `bison -y veriflog.y`*, puis compilation C et éditions de liens* `gcc -o veriflog y.tab.c`*, il ne reste plus qu'à lancer l'exécutable* `veriflog` *obtenu :*

```
1&0|((0)|0|1)
Expression reconnue
```

En relançant à nouveau `veriflog` *:*

```
1&0|((0)|0|1|a)parse error

Expression non reconnue
```

L'analyseur syntaxique généré tente, de reconnaitre un mot du langage défini par la grammaire. Il exécute les instructions correspondantes à chaque règle reconnue. Dans cette exemple, il n'y a aucune action associée aux règles. L'analyseur termine sur la fin de fichier (EOF) de l'entrée standard (CTRL-D pour le terminal).

Au cœur du source C `y.tab.c` généré par `bison`, la fonction C : `int yyparse()` d'analyse syntaxique permet de retourner la valeur 1 en cas d'erreur syntaxique, 0 sinon. La fonction principale : `int main()` appelle `yyparse()` qui va appeler `yylex()` itérativement au fur et à mesure de la reconnaissance des règles de grammaires.

En cas d'erreur de syntaxe, `yyparse()` fait appel à `yyerror(char *)` pour informer l'utilisateur puis `yyparse()` retourne 1.

L'option `-y` de `bison` permet de générer un fichier nommé `y.tab.c`, comme en `yacc`. Sans cette option, le fichier généré se nommerait `veriflog.tab.c`.

3.3.2 Syntaxe et sémantique des sources yacc

Architecture

Un source yacc comprend 3 parties séquentielles :

— une partie déclaration contenant des déclarations C contenues entre `%{` et `%}`, et des déclarations spécifiques à yacc.

— Délimitée par %% au début, une partie constituées de règles de grammaire et des actions associées à la reconnaissance de chaque règle. C'est la partie centrale du source yacc qui définit l'analyseur syntaxique.

— Délimitée par %% au début, une partie de fonctions C définies par l'utilisateur. Dans le cas de Bison, on doit définir au moins trois fonctions : le main(), yyerror() et yylex(). Remarquons que ces fonctions peuvent être définies dans un autre fichier qui sera lié après compilation. Dans le cas de Yacc, une librairie liby.a contient des définitions par défaut de ces trois fonctions.

Les règles de grammaires yacc

Une règle yacc se présente de la façon suivante : un symbole non terminal, le caractère " :", une séquence de symboles (terminaux (jetons) ou non terminaux) et de blocs d'actions `{...}`, terminé par un" ;".

L'espace, la tabulation et le retour à la ligne ne sont pris en compte que comme séparateurs. La règle doit commencer en début de ligne et terminer par un " ;".

Symboles terminaux (jetons) et non terminaux

Les symboles terminaux ou jetons sont représentés par un entier (int) retourné par la fonction d'analyse lexicale yylex(). Les jetons peuvent être

non nommés comme '&', '1' dans l'exemple précédent. En fait dans cet exemple, tous les jetons étaient non nommés.

ou bien nommés . Dans ce cas, yylex() et yyparse() doivent partager une définition (`#define`) commune de ces jetons. La manière la plus simple consiste à

1. les déclarer, dans la première partie du source yacc à l'aide du déclarateur yacc : `%token NAME`. Par convention, les noms de jeton sont en majuscules.
2. Générer un fichier y.tab.h contenant les `#define` correspondant grâce à l'option -d du compilateur yacc.
3. Inclure ce fichier dans la partie définition du source lex.

Bien entendu, si l'on n'utilise pas lex, cette dernière opération est inutile.

Dans l'exemple précédent, on remplace les jetons non nommés '0' et '1' par ZERO et UN.

```
%token UN ZERO
%%
...
        |       ZERO
        {}
        |       UN
        {}
        ;
%% /* debut des fonctions C */
int yylex() {       /* analyseur lexical filtrant les blancs */
  int c;
  while(((c=getchar())==' ') || (c=='\t'))
    ;
  if (c=='0')
    return ZERO;
  else
    if (c=='1')
      return UN;
    else
      return (c);
}
```

Si l'on regarde le fichier `y.tab.h` après la commande `bison -yd ...`, on observe :

```
#define UN      258
#define ZERO    259
```

Rappelons que `yylex()` généré par `lex` retourne 0 en fin de fichier. Les caractères ascii ont un numéro de jeton égal à leur code ascii ! Enfin, un jeton spécial `error` est réservé pour la gestion des erreurs.

Les symboles non terminaux sont conventionnellement écrits en minuscules (`expr, statement,` ...).

Exercice 4 Ecrire le source `yacc` de vérification du langage de Dyck.

Partie droite de règle

Les différentes productions associées au même non-terminal seront séparées par une barre verticale "|". Une partie droite peut être vide afin d'indiquer une epsilon-production. Par exemple :

```
list    :       /* epsilon-production */
        |       list ',' stat
        ;
```

Les différentes productions pourraient cependant être écrites séparément (l : ; l :l ',' s ;). La récursivité à gauche et à droite est permise dans les règles yacc, cependant il est fortement recommandé d'écrire des grammaires récursives à gauche pour optimiser le fonctionnement de l'analyseur.

Valeur sémantique ou attribut

Associée à chaque symbole, terminal ou non, une **valeur sémantique** (attributs des grammaires attribuées) est définie automatiquement par `yacc`. Le type `YYSTYPE` (YY Semantic TYPE) par défaut de cet attribut est entier (`int`) mais peut être défini de deux façons :

- si l'on a besoin que d'un seul type sémantique pour tous les symboles de la grammaire, il suffit de définir YYSTYPE par une macro dans les déclarations C : `#define YYSTYPE double` ; attention à répéter cette macro également dans le source lex avant l'inclusion de `y.tab.h` sinon lex utilisera le type par défaut `int`.
- si l'on a besoin de plusieurs types sémantiques pour différents symboles, par exemple `int` et `float`, on utilisera la déclaration yacc `union`. Par exemple,

  ```
  %union {
    int typeEntier;
    float typeFlottant;
  }
  ```

  dans la section déclaration, redéfinit `YYSTYPE` comme suit :

```
typedef union {
  int typeEntier;
  float typeFlottant;
} YYSTYPE;
```

La variable globale `yylval` est l'attribut que `yylex()` peut affecter aux jetons. Ainsi, par exemple, toutes les littéraux entiers seront associés au même jeton `LITINT` mais auront une valeur sémantique `yylval.typeEntier` différente correspondant à leur valeur. De même pour les littéraux flottants qui correspondront au jeton `LITFLOT` mais qui différeront sur `yylval.typeFlottant`. La déclaration de `yylval` dans `y.tab.h` est de la forme : `extern YYSTYPE yylval;`.

Actions

N'importe quelle instruction C peut apparaître dans un bloc d'actions. De plus, `yacc` admet des ations spécifiques permettant d'utiliser les attributs. L'attribut associé à la partie gauche de la règle de production courante est nommé `$$`, tandis que l'attribut du nième élément de la partie droite est nommé `$n`.

Exemple 23
Un exemple d'utilisation de ces attributs est l'amélioration du vérificateur de G_B en un interpréteur d'expression booléenne :

```
%{ /* interlog.y  */
#include  <stdio.h>
#define YYSTYPE int             /* inutile */
int yylex(void);  void yyerror(char *s);
%}
%%
ligne   :       expr '\n'       {printf("\nRésultat : %d\n",$1);}
        ;
expr    :       '(' expr ')'    {$$ = $2;}
        |       expr '|' expr   {$$ = $1 || $3;}
        |       expr '&' expr   {$$ = $1 && $3;}
        |       '!' expr        {$$ = ! $2;}
| '0'             {$$ = 0;}
        |       '1'             {$$ = 1;}
        ;
%%
int yylex(void) {
  int c;
  while(((c=getchar())==' ') || (c=='\t'))
    ;
  return c;
}
void yyerror(char *s) {
  fprintf(stderr,"%s\n",s);
}
int main(void){
  printf("Veuillez entrer une expression booléenne S.V.P. : ");
  return yyparse();
}
```

Un exemple d'utilisation de cet interprète :

```
0|!0&1
Résultat : 1
!1&0
Résultat : 1
```

Le dernier résultat n'est pas cohérent en logique mais est le résultat de la non définition de priorité d'opérateur dans notre source `bison`.

Actions à l'intérieur de la partie droite

Un bloc d'actions peut apparaître au début et/ou au milieu de la partie droite de la règle. Ces actions peuvent faire référence aux attributs associés aux symboles les précédants. Ces actions sont exécutées après la reconnaissance

des symboles les précédant et avant la reconnaissance des symboles suivants. Attention, un bloc d'action intermédiaire est comptabilisé comme un autre symbole dans la numérotation des attributs `$$n`. En effet, un bloc intermédiaire est lui-même associé à un attribut `$n` correspondant à sa position dans la partie droite. A l'intérieur du bloc intermédiaire, la valeur de l'attribut associé à ce bloc peut être défini en affectant l'attribut `$$`. Attention, `$$` référence l'attribut de bloc et non pas l'attribut de la partie gauche de règle ! Ce dernier ne peut être défini que par une action de fin de règle. Le type d'un bloc intermédiaire ne peut qu'être explicitement donné lors de son utilisation par : `$<typeBloc>$` ou `$<typeBloc>n`. Le typeBloc pouvant être n'importe lequel des types définis par `YYSTYPE`. Prenons l'exemple du langage C, dans lequel un bloc d'instructions est composé de déclarations (optionnelles) suivies d'instructions, le tout entre accolades :

```
bloc:  '{' {initPourDeclarations();} decls insts '}'
    |  '{' insts '}'
    ;
```

Dans cet exemple, le symbole non terminal `decls` a un attribut référencé par `$3`.

Actions prédéfinies

$$ attribut du non terminal en partie gauche de règle ;

$n attribut associé au n ième composant de la partie droite ;

$<typeAutre>n permet de spécifier un autre type que le type par défaut du n ième composant ;

YYABORT retourne immédiatement de yyparse avec un résultat 1 (erreur) ;

YYACCEPT retourne immédiatement de yyparse avec un résultat nul 0 ;

YYBACKUP(jeton, valeurAttribut) dépile un jeton de l'automate ...

yychar variable entière contenant le jeton de prévision courant ;

YYEMPTY valeur stockée dans yychar quand il n'existe pas de jeton de prévision ;

YYERROR provoque une erreur de syntaxe immédiate ;

YYRECOVERING variable valant 1 si on est dans une récupération d'erreur, 0 sinon ;

yyclearin supprime le le jeton de prévision courant ;

yyerrok force le retour de la récupération d'erreur vers l'état normal de l'analyseur syntaxique. Il faut être sur d'être à un bon "endroit" du flot de jeton pour appeler cette fonction. Dans les interpréteurs ligne à ligne, un bon endroit se situe après le retour ligne.

La partie déclaration

Le type `YYSTYPE` des attributs doit être défini par la déclaration `%union` :

```
%union {
  int typeEntier;
  float typeFlottant;
}
```

Les jetons nommés doivent être déclarés dans cette section ainsi que le type de leur attribut par une déclaration du genre : `%token <typeFlottant> LITTERALFLOTTANT`. Il est inutile de spécifier le code numérique du jeton, car `yacc` s'en charge, ce qui évite des erreurs de conflits.

En cas de types multiples des attributs, les symboles non terminaux doivent être tous typés par une déclaration : `%type <typeFlottant> nonterminal1 nonterminal2 ....`

Par défaut, l'axiome de la grammaire est le premier non terminal rencontré dans la partie des règles. On peut définir explicitement l'axiome par la déclaration : `%start nonterminal`.

Associativité et priorité des opérateurs

Dans la partie déclaration, on peut définir des conventions de **priorité** d'opérateurs et les règles définissant leur type d'**associativité**. Rappelons qu'un opérateur binaire infixe * est associatif à gauche ("left") lorsque $x*y*z = (x*y)*z$ et associatif à droite ("right") lorsque $x*y*z = x*(y*z)$. Lorsqu'un opérateur est associatif à gauche et à droite, il faudra choisir l'une des deux associativités pour indiquer l'ordre d'évaluation des expressions. Si un opérateur est non associatif, c'est-à-dire $x*y*z$ n'est pas défini, il faudra également l'indiquer à yacc. La déclaration de l'associativité à gauche est effectuée par : `%left JETONOP1 JETONOP2 JETONOP3 ...` où `JETONOPi` est un jeton nommé ou non d'opérateur. On utilise de même `%right` et `%nonassoc` pour l'associativité à droite et la non associativité. Dans ce dernier cas, si l'analyseur trouve $x*y*z$ alors que * est non associatif, une erreur de syntaxe sera générée.

La priorité des opérateurs, les uns par rapport aux autres, peut être définie simplement par l'ordre des définitions des associativités des opérateurs, du **moins prioritaire au plus prioritaire**. Enfin, une priorité différente de celle de l'opérateur en cours de reconnaissance peut être affectée à une partie droite de règle en ajoutant `%prec JETONVIRTUEL` à la fin de la règle. Ainsi, l'opérateur obtiendra, pour cette règle la priorité (précédence) du `JETONVIRTUEL` qui aura du être déclaré.

Exemple 24

```
%nonassoc '<' '>' EGAL DIFFERENT SUPEGAL INFEGAL
%left '+' '-'
%left '*' '/'
%right MOINSUNAIRE
%right '^'
...
expr : ...
     | expr '-' expr              {/* priorité normale du moins binaire */}
     | '-' expr %prec MOINSUNAIRE {/* priorité spéciale du moins unaire */}
```

Ce type de précédence variable pour le même jeton lexical est nécessaire lorsqu'un opérateur est utilisé dans des emplois différents. On peut prendre comme autre exemple l'opérateur * du C++, utilisé pour la multiplication et le déréférencement d'un pointeur : `*ptrInt * 2`.

L'automate à pile choisit l'opération `Shift` ou `Reduce` en comparant la priorité de la règle courante avec celle du jeton de prévision. Si le jeton est plus prioritaire alors un `Shift` est effectué, sinon un `Reduce` est effectué. La priorité d'une règle est la priorité de son jeton le plus à droite. Les jetons sans priorité explicite sont considérés comme ayant une priorité minimale.

Interface avec lex

`yyparse()` appelle itérativement `yylex()` jusqu'à ce que celui-ci retourne un jeton inférieur ou égal à 0. Les noms de jetons nommés peuvent être partagés par l'intermédiaire du fichier `y.tab.h` qui est automatiquement généré lorsqu'on utilise l'option `-d` de `yacc`. La valeur sémantique (attribut) d'un jeton sera passée de `lex` à `yacc` par l'intermédiaire de la variable `yylval` qui est de type `YYSTYPE`.

Débogage

Afin de déboguer l'analyseur syntaxique, il suffit de positionner la variable `yacc` prédéfinie `yydebug` à 1 avant l'appel à `yyparse()` ou pendant son exécution .

Makefile

```
YACC=bison
YACCFLAGS=-ydtv
#-y a la yacc : y.tab.c; -d genere y.tab.h; -t debogage possible; -v verbose
.y:
        @echo debut $(YACC)-compil : $<
        $(YACC) $(YACCFLAGS) $<
        @echo debut compil c avec edition de liens de y.tab.c
        $(CC) $(CFLAGS) -o $* y.tab.c
        @echo fin $(YACC)-compil de : $<
        @echo Vous pouvez executer : $*
```

3.3.3 Un exemple complet : une calculette

Les sources `lex calc.l` et `yacc calc.y` définissent une calculette interprétant des expressions arithmétiques décimales. Voici le source `lex` :

```
%{ /* calc.l */
#define YYSTYPE double /* ATTENTION AUX 2 MACROS dans lex et yacc */
#include "y.tab.h"      /* JETONS crees par yacc et definition de yylval */
#include <stdlib.h>/* pour double atof(char *) */
#include <stdio.h>/* pour printf */
%}
chiffre         ([0-9])
entier          ({chiffre}+)
```

```
%option noyywrap
  /* pas de continuation sur un autre fichier */
%%
[ \t]+          {/* filtrer les blancs */}
{entier}|{entier}\.{chiffre}*|{chiffre}*\.{entier} {
                  /* laisser l'accolade à la ligne precedente */
                  yylval=atof(yytext);return (LITFLOT);
                }
sin             { return(SIN); }
cos             { return(COS); }
exp             { return(EXP); }
ln              { return(LN); }
pi              { return(PI); }
exit|quit       { return (QUIT); }
aide|help|\?    { return (HELP); }
.|\n            { return yytext[0]; /* indispensable ! */}
%%
```

Voici le source `yacc` :

```
%{                              /* calc.y  */
#include <math.h>
int errSemantiq=0;              /* vrai si erreur sémantique : */
#define DIVPAR0 1               /* division par 0 */
#define LOGNEG  2               /* logarithme d'un négatif */
#define YYSTYPE double
int yylex(void);void yyerror(char *s);
%}
                                /* définition des jetons */
%token LITFLOT SIN COS EXP LN PI QUIT HELP
                                /* traitement des priorités */
%left '+' '-'
%left '*' '/' '%'
%right MOINSUNAIRE
%right '^'
%%

liste   :       {/* chaine vide sur fin de fichier Ctrl-D */}
        |       liste ligne {}
        ;
ligne   :       '\n'            {/* ligne vide : expression vide */}
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       expr '\n'       {
          if (!errSemantiq)
            printf("Résultat : %10.2f\n",$1); /* 10 car dont 2 décimales */
          else if (errSemantiq==DIVPAR0){
            printf("Erreur sémantique : division par 0 !\n");
            errSemantiq=0; /* RAZ */
          }
          else {
            printf("Erreur sémantique : logarithme d'un négatif ou nul !\n");
            errSemantiq=0; /* RAZ */
          }
        }
        |       QUIT '\n'       {return 0; /* fin de yyparse */}
        |       HELP '\n'       {
        printf("        Aide de la calculette\n");
        printf("        ====================\n");
        printf("Taper une expression constituée de nombres, d'opérations,\n");
        printf("      de fonctions, de constantes, de parenthèses puis taper <Entrée> \n");
        printf("Ou taper une commande suivie de <Entrée>\n\n");
        printf("Syntaxe des nombres : - optionnel, suivi de chiffres, \n");
```

```
        printf("       suivi d'un . optionnel, suivi de chiffres \n");
        printf("Opérations infixes : + - * / ^ %% (modulo) \n");
        printf("Fonctions prédéfinies : sin(x) cos(x) exp(x) ln(x)\n");
        printf("Constantes prédéfinies : pi\n");
        printf("Commandes : exit ou quit pour quitter la calculette\n");
        printf("            aide ou help ou \? pour afficher cette aide\n");
        }
        ;
expr    :       '(' expr ')'    {$$ = $2;}
        |       expr '+' expr   {$$ = $1 + $3;}
        |       expr '-' expr   {$$ = $1 - $3;}
        |       expr '*' expr   {$$ = $1 * $3;}
        |       expr '/' expr   {
          if ($3!=0)
            $$ = $1 / $3;
          else
            errSemantiq=DIVPAR0; /* par défaut $$=$1 */
        }
        |       expr '^' expr   {$$ = pow($1,$3);}
        |       expr '%' expr   {
          if ($3!=0) $$ = fmod($1,$3);
          else errSemantiq=DIVPAR0; /* par défaut $$=$1 */
        }
        |       '-' expr %prec MOINSUNAIRE      {$$ = - $2;}
        |       SIN '(' expr ')'{$$ = sin ( M_PI/180*$3 );}
        |       COS '(' expr ')'{$$ = cos ( M_PI/180*$3 );}
        |       EXP '(' expr ')'{$$ = exp($3);}
        |       LN '(' expr ')' {
          if ($3>0) $$ = log($3);
          else errSemantiq=LOGNEG; /* $$=$1 ... */
        }
        |       PI              {$$ = M_PI;}
        |       LITFLOT         {$$ = $1;}
        ;
%%
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(void){yydebug=0; return yyparse();}
```

3.3.4 Bison (version 2.3) et analyseur C++

On peut utiliser le langage C++ dans bison, dans la partie action. On obtient alors un analyseur écrit en C et C++ : `yyparse()` est une fonction C, et les actions peuvent utiliser des classes ...

Une autre approche consiste à concevoir un analyseur syntaxique totalement C++ : il est alors nécessaire d'effectuer un nombre assez important de modifications !

Il faut utiliser un squelette ("skeleton") de parseur C++ nommé `/usr/share/bison/lalr1.cc` :

— soit en utilisant l'option bison `-skeleton=lalr1.cc` ;

— soit en utilisant la directive `%skeleton "lalr1.cc"`

Grâce à ce squelette, bison va créer différents objets :

1. un espace de nom `yy` dans lequel vont apparaître :
2. la classe `parser` qui contient la méthode `int parse()` ;
3. dans cette classe, le type `semantic_type` qui est l'union des types sémantiques possibles. Attention, une union ne peut contenir de classe (string, map, ...) : il faut donc y mettre un **pointeur sur classe** et utiliser l'allocation dynamique (new) ;
4. dans cette classe, le type `token::token_type` qui est une énumération des jetons nommés (257, 258, ...) ;
5. dans cette classe, le type `location_type` qui permet de localiser les erreurs ;

Attention à la déclaration indispensable de la fonction `yylex` qui doit se situer après la définition `%union`. Après exécution de bison sur le source, on obtient les fichiers suivants :

— location.hh et position.hh définissent des classes de position dans le fichier ;

— stack.hh définit la pile de l'automate ;

— y.tab.h et y.tab.c définissent l'espace de nom et la classe parser ; Il est utile d'observer le fichier y.tab.h !

Voici un exemple minimal d'un analyseur en C++ nommé `parserminiC++.y` :

Exemple 25

```
%{
#include <iostream>
using namespace std;
%}

%skeleton "lalr1.cc"
%union{
  int i;
}

%{ /* A DECLARER ABSOLUMENT APRES L'UNION */
  yy::parser::token_type yylex (yy::parser::semantic_type* pyylval);
%}
%token <i> CHAR

%%
s : CHAR  '\n' {cout<<endl<<"Vous avez tapé le char : "<<$1<<endl;}
  ;
%%

yy::parser::token_type yylex (yy::parser::semantic_type* pyylval){
  pyylval->i=getchar();
  if (pyylval->i=='\n'){ /* \n */
    return yy::parser::token_type('\n');
  }
  if (pyylval->i==-1){ /* EOF == -1 pour getchar()*/
    return yy::parser::token_type(0); /* EOF==0 pour flex ! */
  }
  else return yy::parser::token::CHAR; // un peu long ...
}
void yy::parser::error(yy::location const& loc, std::string const& s){
  cout<<endl<<s<<endl;
}
int main(){
  yy::parser* pparser=new yy::parser(); /* instance */
  int i= pparser->parse(); /* lancer l'analyse */
  if (i==0){
    cout<<"Syntaxe correcte"<<endl;
  } else {
    cout<<"Syntaxe incorrecte"<<endl;
  }
  return i;
}
```

Aprés compilation : `bison -ydtv parserminiC++.y` *puis* `g++ -o mini y.tab.c`*, on obtient l'exécutable* `mini` *dont l'utilisation suit :*

```
>mini
a

Vous avez tapé le char : 97
Syntaxe correcte
>mini
bb

syntax error
Syntaxe incorrecte
```

Pour plus de détails, notamment pour l'utilisation de flex, nous allons étudier un second exemple permettant de maintenir une table de variables entières. Ces variables seront affectées grâce à un interprèteur et seront mémorisées dans une map C++. Suit un exemple de fonctionnement de l'interprète :

Exemple 26

```
>affect
b=5
a=8
b=3
set
a --> 8
b --> 3
a=0
set
a --> 0
b --> 3
quit
Syntaxe correcte
```

Voici le source bison affect.y+ :

```
%{
#include <iostream>
#include <string>
#include <map>
using namespace std;
/* redéfinition du prototype de la fonction yylex qui devra être déclarée dans
   le source flex et dans le source bison */
#define YY_DECL yy::parser::token_type yylex (yy::parser::semantic_type* pyylval)
%}

%skeleton "lalr1.cc"
%union{
  int i;
  string *ps;
}

%{
/* A DECLARER ABSOLUMENT APRES L'UNION (ne sera pas dans le y.tab.h)*/
YY_DECL;
/* tableau des affectations */
map <string, int>* tvar=new map<string, int>();
%}
%token END      0 "end of file"
%token SET QUIT
%token <ps> ID
%token <i> VALEUR
%%
liste : { /* epsilon */}
      | liste ligne {}
      ;
ligne : SET '\n' {
  map <string, int>::iterator j = tvar->begin(); // attribut de liste !
  while (j != tvar->end()){
    cout << j->first<<" --> "<<j->second << endl;
    ++j;
  }
}
     | ID '=' VALEUR '\n' {(*tvar)[*$1]=$3;}
     | '\n' {}
     | QUIT '\n' {return 0;}
     ;
%%
void yy::parser::error(yy::location const& loc, std::string const& s){
  cout<<endl<<s<<endl;
}
int main(){
```

```
  yy::parser* pparser=new yy::parser(); /* instance */
  /* pparser->set_debug_level(1); /* ancien YYDEBUG=1; */
  int i= pparser->parse(); /* lancer l'analyse */
  if (i==0){
    cout<<"Syntaxe correcte"<<endl;
  } else {
    cout<<"Syntaxe incorrecte"<<endl;
  }
  return i;
}
```

Et voici le source flex affect.l+ :

```
%{
#include "y.tab.h"
// déclaration de yylex
YY_DECL;
// pour éviter de retourner 0 à la fin (0 n'est pas un token) ! pas de ;
#define yyterminate() return yy::parser::token::END
%}
/* evite la definition de yywrap() */
%option noyywrap
%%
set    {return yy::parser::token::SET;}
quit   {return yy::parser::token::QUIT;}
[ \t]+ {/* filtrer*/}
[a-zA-Z][a-zA-Z0-9]* {pyylval->ps=new string(yytext);return yy::parser::token::ID;}
[0-9]+  {pyylval->i=atoi(yytext);return yy::parser::token::VALEUR;}
.|\n  {return yy::parser::token_type(yytext[0]);}
%%
```

Enfin l'entrée de makefile :

```
affect : affect.l+ affect.y+
    bison -ydtv affect.y+
    flex affect.l+
    g++ -o affect y.tab.c lex.yy.c
```

Il resterait beaucoup à dire sur l'utilisation de C++ (localisation, renommages, ...). Pour aller plus loin, voir le manuel bison.

3.4 Analyse ascendante par automate à pile

Nous allons étudier l'analyse ascendante et plus particulièrement l'analyse LALR utilisée dans yacc. Rappelons que, partant d'un mot (flot de jetons), on essaie de construire l'arbre de dérivation associé. Cette construction va se faire depuis les feuilles (jetons) en remontant jusqu'à la racine (l'axiome). De plus, on va construire une dérivation droite (Rightmost) et à l'envers ! Les grammaires pouvant être analysées par un analyseur LR doivent, bien entendu, avoir certaines propriétés comme la non ambiguïté.

Prenons un exemple simple pour illustrer le fonctionnement de l'automate à pile.

Exemple 27
Soit la grammaire $G = (\{1, 2, 3, +\}, \{E\}, R, E)$ *avec les règles de* R *suivantes :*

$$E \to 1|2|3|E + E$$

Considérons le mot d'entrée 1+2+3$. L'analyse du mot commence sur le 1 (Left to right scanning). Après avoir empilé (Shift) ce symbole, la règle $E \to 1$ *est appliquée et on empile E. Arrivé sur le +, l'analyseur empile ce symbole car il ne peut pas appliquer de règle. Le 2 est ensuite reconnu comme partie droite de* $E \to 2$*. On empile donc E et on s'aperçoit qu'on peut alors réduire (Reduce) le mot sur la pile (E+E) en appliquant la règle* $E \to E + E$*. La pile ne contient donc plus que E. En continuant le même procédé, on reconnait les productions* $E \to 3$ *puis* $E \to E + E$*. On a donc la dérivation droite, obtenue à l'envers :* $E \overset{1}{\Rightarrow}_{E \to E+E} E + E \overset{1}{\Rightarrow}_{E \to 3} E + 3 \overset{1}{\Rightarrow}_{E \to E+E} E + E + 3 \overset{1}{\Rightarrow}_{E \to 2} E + 2 + 3 \overset{1}{\Rightarrow}_{E \to 1}$.

Remarquons que cette grammaire est ambiguë et qu'on a décrit un analyseur déterministe qui choisit d'évaluer 1+2 en premier et non pas 2+3. Cet analyseur choisit l'action Reduce sur un conflit Shift/Reduce. `Yacc`*, au contraire, privilégie toujours le* `Shift` *sur le* `Reduce`*, ce qui lui permet d'associer naturellement le* `else` *au* `if` *le plus proche ! Mais ceci entraîne l'évaluation des opérateurs de droite à gauche si aucune priorité n'est définie !*

3.4.1 Fonctionnement de l'automate à pile en analyse ascendante LR

Définition 4 *Un manche d'un mot (pas forcément terminal) $m = \alpha\beta\gamma$ est un couple constitué :*
- *d'une production $X \to \beta$,*
- *d'une position p dans m telle que $m[p, p + |\beta|[= \beta$;*

ayant la propriété suivante : $S \overset{*}{\Rightarrow}_d \alpha X\gamma \overset{1}{\Rightarrow}_d m = \alpha\beta\gamma$.

Dans l'exemple 27 précédent, le mot 1+2+3 ne possède qu'un manche ($E \to 1,1$). En effet, ni ($E \to 3,5$), ni ($E \to 2,3$) est un manche car ni 1+2+E, ni 1+E+3 ne dérive de E par une dérivation droite. Par contre, E+E+3 possède deux manches : ($E \to E + E,1$) et ($E \to 3,5$). On peut donc choisir entre les deux réductions possibles. Dans l'exemple 27, nous avions choisi de réduire sur la position la plus à gauche de façon à réduire dès qu'un manche est situé sur la pile. On aurait pu empiler + puis E au dessus de E+E puis réduire par deux fois E+E en E. Nous avions choisi de privilégier la réduction (Reduce) sur le décalage (Shift) dans ce conflit Shift/Reduce.

Malheureusement, l'identification du manche n'est pas toujours aussi simple que dans l'exemple 27. Il peut exister d'autres types de conflits Reduce/Reduce lorsque deux manches sont réductibles. Pour limiter ces conflits d'action, la table d'analyse ainsi que la pile vont utiliser des états entiers correspondant à la configuration courante, c'est-à-dire à ce qui a été reconnu jusqu'alors.

Définition 5 *La pile d'un analyseur LR est une structure Dernier Entré Premier Sorti (LIFO) de couples (s,e) où $s \in V \cup \{\$\}$ est un symbole et $e \in \mathbb{N}$ est un état entier. L'état courant de l'analyseur est l'état situé au sommet de la pile.*

Définition 6 *La table d'analyse d'un analyseur LR est constitué d'une partie Action et d'une partie Successeur.*
- *La table d'action est un tableau à deux entrées : les différents états sur les lignes, les terminaux et $ sur les colonnes. On note une case de cette table par $Action[e, x]$. Une action d'un analyseur LR peut être :*
  - *Décaler (Shift) le symbole courant du flot d'entrée sur la pile (empiler) avec un état e. Cette action est notée : Se.*
  - *Réduire (Reduce) par une production $X \to \alpha$. Cela consiste à dépiler α (à l'envers) de la pile et à le remplacer par X et l'état correspondant dans la table Successeur, c'est à dire $Successeur[sommet(Pile)[2], X]$. Cette action est notée : $R(X \to \alpha)$.*
  - *Accepter le mot d'entrée et terminer l'analyse. Cette action est notée : Accepter.*
  - *Générer un message d'erreur de syntaxe et terminer l'analyse. Cette action n'est pas notée explicitement : toutes les cases vides de la table Action représentent des actions Erreur.*
- *La table des successeurs est un tableau à deux entrées : les différents états sur les lignes, les non terminaux sur les colonnes. On note une case de cette table par $Successeur[e, X]$. Cette table ne sert qu'à indiquer le nouvel état courant après une réduction. Là aussi, toutes les cases vides de la table Successeur représentent des erreurs.*

Avant de voir les algorithmes de construction de ces tables, regardons le fonctionnement de l'analyseur. L'analyse d'un mot du flot d'entrée est décrit dans l'algorithme 16.

Algorithme 16 : Fonctionnement de l'automate

Données : Une table d'analyse $Action[Etat, V_T \cup \{\$\}], Successeur[Etat, V_N]$, un flot de jetons terminé par $
Résultat : Erreur ou Succès
Pile=construirePileVide() *// contenu : (symbole, état)*
empiler(Pile,($,0)) *// initialisation*
jeton=lireFlot() *// jeton courant du flot*
tant que *vrai* **faire**
  etatCourant=sommet(Pile)[2] *// projection sur l'état*
  exécuter $Action[etatCourant, jeton]$ *// Shift, Reduce, Erreur ou Accepter*

Pour illustrer le fonctionnement de l'algorithme 16, prenons un exemple simple d'une grammaire de Dyck à un couple de parenthèses.

Exemple 28
Soit la grammaire $G_d = (\{a, b\}, \{S\}, R, S)$ avec les règles de R suivantes :

$$S \to SaSb|\varepsilon$$

Le calcul des tables de cette grammaire fournit le résultat suivant :

	Action			*Successeur*
	a	*b*	*$*	*S*
0	$R(S \to \varepsilon)$	$R(S \to \varepsilon)$	$R(S \to \varepsilon)$	*1*
1	*S2*		*Accepter*	
2	$R(S \to \varepsilon)$	$R(S \to \varepsilon)$	$R(S \to \varepsilon)$	*3*
3	*S2*	*S4*		
4	$R(S \to SaSb)$	$R(S \to SaSb)$	$R(S \to SaSb)$	

Examinons l'analyse du mot abaababb$:

Pile	*Flot d'entrée*	*Action*
$0	*abaababb$*	$R(S \to \varepsilon)$
$0S1	*abaababb$*	*S2*
$0S1a2	*baababb$*	$R(S \to \varepsilon)$
$0S1a2S3	*baababb$*	*S4*
$0S1a2S3b4	*aababb$*	$R(S \to SaSb)$
$0S1	*aababb$*	*S2*
$0S1a2	*ababb$*	$R(S \to \varepsilon)$
$0S1a2S3	*ababb$*	*S2*
$0S1a2S3a2	*babb$*	$R(S \to \varepsilon)$
$0S1a2S3a2S3	*babb$*	*S4*
$0S1a2S3a2S3b4	*abb$*	$R(S \to SaSb)$
$0S1a2S3	*abb$*	*S2*
$0S1a2S3a2	*bb$*	$R(S \to \varepsilon)$
$0S1a2S3a2S3	*bb$*	*S4*
$0S1a2S3a2S3b4	*b$*	$R(S \to SaSb)$
$0S1a2S3	*b$*	*S4*
$0S1a2S3b4	*$*	$R(S \to SaSb)$
$0S1	*$*	*Accepter*

Ce qui donne la dérivation droite suivante : $S \overset{1}{\Rightarrow} SaSb \overset{1}{\Rightarrow} SaSaSbb \overset{1}{\Rightarrow} SaSabb \overset{1}{\Rightarrow} SaSaSbabb \overset{1}{\Rightarrow} SaSababb \overset{1}{\Rightarrow} Saababb \overset{1}{\Rightarrow} SaSbaababb \overset{1}{\Rightarrow} Sabaababb \overset{1}{\Rightarrow} abaababb$

3.4.2 Algorithmique

Nous allons décrire comment calculer les tables d'analyses pour des grammaires LR(1), c'est-à-dire avec un symbole de prévision. Il existe plusieurs méthodes de construction dépendant de la complexité de la grammaire et de l'efficacité de l'analyseur, notamment en ce qui concerne la taille des tables. La méthode SLR, "Simple LR", permet de construire très efficacement des tables d'analyse assez petites. Malheureusement, certaines constructions syntaxiques, peu nombreuses dans les langages de programmation, ne peuvent être gérées par cette méthode. D'autres méthodes existent, dont la méthode LALR de yacc, résolvant certains problèmes de SLR au prix d'une taille plus importante des tables. Enfin, il existe une méthode dite canonique qui assure la reconnaissance de toute grammaire LR(1) mais à un cout prohibitif.

Nous nous contenterons ici de décrire la méthode SLR en conseillant le livre [1] pour ceux qui souhaiteraient en savoir plus.

Construction de la collection canonique SLR

Définition 7 *Un item LR(0), ou SLR, ou plus simplement item, d'une grammaire* $G = (V_T, V_N, R, S)$ *est un couple constitué d'une production de R et d'une position dans la partie droite de celle-ci. La position est représentée par un point '.' dans la parte droite.*

Soit la grammaire $G_d = (\{a, b\}, \{S\}, R = \{S \to SaSb|\varepsilon\}, S)$. L'ensemble des items de G est $Items(G) = \{S \to .SaSb, S \to S.aSb, S \to Sa.Sb, S \to SaS.b, S \to SaSb., S \to \varepsilon.\}$. Un item représente ce qui a déjà été reconnu (à gauche du point) lors de l'analyse, et ce qu'il reste à reconnaitre (à droite du point) avant de pouvoir réduire. Avant de construire les tables Action et Successeur, il faut calculer un automate fini déterministe (ou collection canonique), c'est à dire un ensemble d'états reliés par des transitions. Chaque état représente un ensemble d'items correspondant à une situation d'analyse. Ces états sont les états de l'analyseur LR.

Définition 8 *Une grammaire augmentée G' d'une grammaire* $G = (V_T, V_N, R, S)$ *est obtenue par ajout d'un nouvel axiome S' et d'une production* $S' \to S$ *:* $G' = (V_T, V_N \cup \{S'\}, R \cup \{S' \to S\}, S')$

L'ajout de ce "super-axiome" est motivé par l'obtention d'un état initial de l'AFD qui soit une source : on ne peut revenir sur cet état initial. La construction de l'AFD utilise une fonction Fermeture() qui regroupe tous les items auxquels on peut s'attendre dans un état donné. La fonction Fermeture() est décrite dans l'algorithme 17.

Algorithme 17 : Fermeture d'un ensemble d'items

Données : Un ensemble I d'items d'une grammaire augmentée $G = (V_T, V_N, R, S)$
Résultat : Un ensemble d'items
Fermeture(I)=I // *initialisation*
pour chaque *item non marqué* $j = \alpha.X\beta \in Fermeture(I)$ *tel que* $X \in V_N$ **faire**
  marquer j // *on ne traite un item qu'une seule fois*
  pour chaque *production* $X \to \gamma \in R$ **faire**
    $Fermeture(I) = Fermeture(I) \cup \{X \to .\gamma\}$
retourner Fermeture(I)

Le principe de l'algorithme 17 tient en ce que lorsqu'on s'attend à reconnaitre un non terminal X, il faut également s'attendre à reconnaitre toute partie droite de production dont X est la partie gauche.

Exemple 29
Soit la grammaire de Dyck augmentée : $G' = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$. *Calculons les fermetures des ensembles d'items* $\{S' \to .S\}$ *et* $\{S \to Sa.Sb\}$. $Fermeture(\{S' \to .S\}) = \{S' \to .S, S \to .SaSb, S \to \varepsilon.\}$ *et* $Fermeture(\{S \to Sa.Sb\}) = \{S \to Sa.Sb, S \to .SaSb, S \to \varepsilon.\}$.

Pour construire l'AFD des états de l'analyseur, également appelée collection canonique des ensembles d'items LR(0), il faut examiner toutes les transitions possibles d'un état (ensemble d'items) vers un autre par le déplacement du "." d'une position vers la droite. L'algorithme 18 décrit cette construction.

Algorithme 18 : Construction de l'AFD

Données : Une grammaire augmentée $G = (V_T, V_N, R, S')$
Résultat : Un AFD $B = (V, E, D, A, T)$ ou collection canonique
$V = V_T \cup V_N - \{S'\}$ // *les symboles de transition sont les symboles de la grammaire non augmentée*
$E = \{Fermeture(\{S' \to .S\})\}$ // *initialisation de l'ensemble des états*
$D = E$ // *unique état initial*
répéter
  choisir un état non marqué $I \in E$ // *un état est un ensemble d'items*
  marquer I // *on ne traite un état I qu'une seule fois*
  pour chaque $x \in V$ *tel qu'il existe au moins un* $Y \to \alpha.x\beta \in I$ **faire**
    $transition(I, x) = Fermeture(\{Y \to \alpha x.\beta\})$ // *calcul de l'état suivant après reconnaissance de x*
    $E = E \cup transition(I, x)$ // *ajout possible d'un nouvel état*
    $T = T \cup \{(I, x, transition(I, x))\}$ // *ajout d'une nouvelle transition*
jusqu'à *ce que tous les états de E soient marqués*;

Remarquons que l'algorithme 18 ne calcule pas d'états d'arrivée de l'automate. En effet, cet automate ne permet pas de reconnaitre un mot du langage analysé mais sert uniquement à décrire les transitions entre états. Chaque chemin dans l'AFD correspond à un préfixe d'un mot dérivant de l'axiome. Ces préfixes, aussi appelé **préfixes viables**, sont constitués de terminaux et de non terminaux. Ils représentent le **contenu possible de la pile** de l'automate à un instant donné.

Exemple 30
Soit la grammaire de Dyck augmentée : $G' = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$. *Calculons l'automate correspondant :*
$I_0 = Fermeture(\{S' \to .S\}) = \{S' \to .S, S \to .SaSb, S \to \varepsilon.\}$
$I_1 = Fermeture(\{S' \to .S, S \to .SaSb\}) = \{S' \to S., S \to S.aSb\}$
$T = \{(I_0, S, I_1)\}$
$I_2 = Fermeture(\{S \to Sa.Sb\}) = \{S \to Sa.Sb, S \to .SaSb, S \to \varepsilon.\}$
$T+ = \{(I_0, S, I_1), (I_1, a, I_2)\}$
$I_3 = Fermeture(\{S \to SaS.b, S \to S.aSb\}) = \{S \to SaS.b, S \to S.aSb\}$
$T+ = \{(I_0, S, I_1), (I_1, a, I_2), (I_2, S, I_3)\}$
$I_4 = Fermeture(\{S \to SaSb.\}) = \{S \to SaSb.\}$
$I_2 = Fermeture(\{S \to Sa.Sb\}) = \{S \to Sa.Sb, S \to .SaSb, S \to \varepsilon.\}$
$T+ = \{(I_0, S, I_1), (I_1, a, I_2), (I_2, S, I_3), (I_3, b, I_4), (I_3, a, I_2), \}$

Dans cet exemple, les préfixes viables sont : $\varepsilon, S, Sa, SaS, SaSb, SaSaSb, \ldots, SaS(aS)^n b$. La question que l'on se pose est de savoir quand un préfixe situé en pile doit être réduit. Définissons la notion d'item valide pour un préfixe viable.

Définition 9 *Un item $X \to \beta_1.\beta_2$ est valide pour un préfixe $\alpha\beta_1$ d'un mot dérivant de l'axiome si et seulement s'il existe une dérivation droite : $S' \overset{*}{\Rightarrow}_d \alpha X m \overset{1}{\Rightarrow}_d \alpha\beta_1\beta_2 m$ avec $m \in {V_T}^*, X \in V_N, \alpha\beta_1\beta_2 \in V^*$.*

Remarquons que dans le cas où l'item $X \to \beta_1.$ est valide pour le préfixe $\alpha\beta_1$, alors on a un manche qu'il faut réduire. Dans le cas où l'item $X \to \beta_1.\beta_2$ est valide et que β_2 n'est pas vide, il faut décaler. La question est maintenant de savoir quand un item est valide pour un préfixe donné.

Théorème 13 *L'ensemble des items valides pour le préfixe viable $\alpha\beta_1$ est l'ensemble des items atteint par un parcours de l'AFD depuis l'état initial, le long du chemin étiqueté par $\alpha\beta_1$.*

Ainsi, l'automate construit permet de répondre facilement à la question précédente.

Exemple 31
Soit le préfixe viable SaS, les deux items valides sont $S \to SaS.b$ et $S \to S.aSb$. On a donc les deux types de dérivations droites possibles : $S \overset{1}{\Rightarrow} SaSb$ ou bien $S \overset{1}{\Rightarrow} SaSb \overset{1}{\Rightarrow} SaSaSb \overset{*}{\Rightarrow} SaSa\ldots$ Remarquons que le symbole d'entrée suivant (a ou b) permettra de choisir l'état suivant qui correspondra soit à une réduction par $S \to SaSb$ ou bien par $S \to \varepsilon$.

Construction des tables d'analyse SLR

On peut maintenant écrire l'algorithme 19 de construction de la table Action d'analyse SLR.

Algorithme 19 : Construction de la table Action en analyse SLR

Données : Une grammaire augmentée $G = (V_T, V_N, R, S')$, un AFD $B = (V, E, D, A, T)$ ou collection canonique
Résultat : La table d'analyse $Action[E, V_T \cup \{\$\}]$
pour chaque *état* $I_j \in E$ **faire**
  pour chaque *item* $i \in I_j$ **faire**
    suivant *l'item* i **faire**
      cas où $i = S' \to S.$
        ajouter "Accepter" à $Action[I_j, \$]$
      cas où $i = X \to \alpha.a\beta$ *avec* $a \in V_T$ *et* $(I_j, a, I_k) \in T$
        ajouter Shift I_k à $Action[I_j, a]$
      cas où $i = X \to \alpha.$ *et* $i \neq S' \to S.$
        pour chaque $x \in TabSuivants[X]$ **faire**
          ajouter $Reduce(X \to \alpha)$ à $Action[I_j, x]$
      cas où *autres*
        ne rien faire
  pour chaque *case vide* $Action[I_j, x]$ **faire**
    écrire "Erreur" dans $Action[I_j, x]$

Remarquons qu'une seule action Accepter existe qui correspond à la réduction $S' \to S$ de la grammaire augmentée. Une case de la table Action peut contenir plusieurs actions ! On peut obtenir des conflits Shift/Reduce ou Reduce/Reduce. Dans ce cas, la grammaire n'est pas SLR et il sera nécessaire d'utiliser un algorithme de construction de table plus complexe.

Exemple 32
Pour appliquer l'algorithme 19 sur la grammaire de Dyck augmentée $G' = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$, il nous faut calculer les suivants de S : $TabSuivants[S] = \{a, b, \$\}$. On obtient alors la table suivante :

	Action		
	a	*b*	*$*
0	*$R(S \to \varepsilon)$*	*$R(S \to \varepsilon)$*	*$R(S \to \varepsilon)$*
1	*S2*	*Erreur*	*Accepter*
2	*$R(S \to \varepsilon)$*	*$R(S \to \varepsilon)$*	*$R(S \to \varepsilon)$*
3	*S2*	*S4*	*Erreur*
4	*$R(S \to SaSb)$*	*$R(S \to SaSb)$*	*$R(S \to SaSb)$*

Algorithme 20 : Construction de la table Successeur en analyse SLR

Données : Une grammaire augmentée $G = (V_T, V_N, R, S')$, un AFD $B = (V, E, D, A, T)$ ou collection canonique
Résultat : La table d'analyse $Successeur[E, V_N]$
pour chaque *transition* $(I_j, X, I_k) \in T$ *tel que* $X \in V_N$ **faire**
  $Successeur[I_j, X] = I_k$
pour chaque *case vide* $Successeur[I_j, X]$ **faire**
  écrire "Erreur" dans $Successeur[I_j, X]$

On peut maintenant écrire l'algorithme 20 de construction de la table Successeur SLR.

Remarquons qu'il ne peut y avoir de conflit car l'automate est déterministe. La table Successeur permet de déterminer l'état courant après une réduction en fonction de l'état sous-jacent dans la pile.

Exemple 33
L'algorithme 20 sur la grammaire de Dyck augmentée $G' = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$ *fournit la table suivante :*

	Successeur
	S
0	*1*
1	*Erreur*
2	*3*
3	*Erreur*
4	*Erreur*

Efficacité

Théorème 14 *Une grammaire est LR(0) ou SLR si et seulement si sa table Action ne contient aucun conflit.*

Théorème 15 *Un langage est LR(0) ou SLR si et seulement s'il existe une grammaire SLR le générant.*

Différentes grammaires SLR existant pour un même langage, on peut se préoccuper de la "meilleure" en terme d'efficacité. Par exemple, nous avons souvent considérée la grammaire augmentée de Dyck suivante : $G_g = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$. Il existe une autre grammaire SLR engendrant le même langage : $G_d = (\{a, b\}, \{S, S'\}, \{S \to aSbS|\varepsilon, S' \to S\}, S')$.

Exercice 5 Construire les tables d'analyse SLR de G_d. Examiner le fonctionnement de l'analyseur sur le mot *abaababb*\$.

Après construction des tables SLR de cette seconde grammaire, on s'aperçoit qu'elles possèdent un état de plus, mais surtout que la reconnaissance d'un mot nécessite une pile beaucoup plus importante. En effet, la première réduction par $S \to aSbS$ ne peut avoir lieu que très tard par rapport à l'analyseur de la grammaire G_g. La raison principale de cette inefficacité tient en ce que G_d est récursive à droite. Par conséquent, on préférera toujours, quand on a le choix, utiliser des grammaires **récursives à gauche** en analyse ascendante.

3.5 Les conflits et leur résolution par yacc

Des grammaires extrèmement simples et non ambiguës peuvent être non SLR. Par exemple, la grammaire augmentée $G = (\{a, b, c\}, \{S', S, A, B\}, \{S' \to S, S \to Aaa|Bab|aac, A \to a, B \to a\}, S)$ est non SLR. Pour le montrer, commençons à construire l'AFD :
$I_0 = Fermeture(\{S' \to .S\}) = \{S' \to .S, S \to .Aaa, S \to .Bab, S \to .aac, A \to .a, B \to .a\}$
$I_1 = Fermeture(\{S \to a.ac, A \to a., B \to a.\}) = \{S \to a.ac, A \to a., B \to a.\}$
$I_2 = Fermeture(\{S \to aa.c\}) = \{S \to aa.c\}$
$T = \{(I_0, a, I_1), \ldots\}$
$TabSuivants[A] = TabSuivants[B] = \{a\}$
Nous pouvons maintenant construire un morceau de la table Action :

	Action	
	a	...
0	S1	...
1	R($A \to a$),R($B \to a$),S2	...
2	...	...

Quel que soit le mot d'entrée, il commence par aa. La lecture du premier a produit un décalage, puis il existe trois actions possibles : deux réductions différentes et un décalage ! En fait, dans ce cas il faudrait examiner la troisième lettre pour choisir la bonne réduction ou le décalage. Cette grammaire n'est pas LR(1) mais LR(2), par conséquent la méthode SLR ne peut rien (pas plus qu'aucune autre méthode LR(1)).

D'autres méthodes existent pour les grammaires LR(1). En particulier, la méthode LALR de yacc, ou bison, qui construit automatiquement les tables d'analyse. L'option `-v` de bison permet notamment de visualiser les tables d'analyse utilisées. Voici, par exemple, le fichier `.output` obtenu avec la grammaire $G_g = (\{a, b\}, \{S, S'\}, \{S \rightarrow SaSb|\varepsilon, S' \rightarrow S\}, S')$.

```
state 0
    $default    reduce using rule 2 (S)
    S           go to state 1
state 1
    S  ->  S . 'a' S 'b'   (rule 1)
    $           go to state 5
    'a'         shift, and go to state 2
state 2
    S  ->  S 'a' . S 'b'   (rule 1)
    $default    reduce using rule 2 (S)
    S           go to state 3
state 3
    S  ->  S . 'a' S 'b'   (rule 1)
    S  ->  S 'a' S . 'b'   (rule 1)
    'a'         shift, and go to state 2
    'b'         shift, and go to state 4
state 4
    S  ->  S 'a' S 'b' .   (rule 1)
    $default    reduce using rule 1 (S)
state 5
    $           go to state 6
state 6
    $default    accept
```

On retrouve, à quelques détails près, les tables Action et Successeur obtenus dans les exemples 32 et 33.

Conflit Shift/Reduce

Que fait yacc lorsqu'il rencontre des conflits ? Sur conflit Shift/Reduce, yacc avantage **toujours l'action Shift**. L'une des raisons historiques de ce choix concerne les "si alors sinon" imbriqués. Soit la grammaire suivante :

$$G_F = (\{i, t, e, a, b\}, \{S, E\}, R, S)$$

avec les règles de R suivantes :

$$\begin{aligned} S &\rightarrow iEtS|iEtSeS|a \\ E &\rightarrow b \end{aligned}$$

La compilation yacc fournit un analyseur privilégiant le décalage du "else" plutôt que la réduction du iEtS empilé. Voici la partie descriptive fournie par `yacc -v` :

```
state 6
    S  ->  'i' E 't' S .   (rule 1)
    S  ->  'i' E 't' S . 'e' S   (rule 2)

    'e'         shift, and go to state 7
    'e'         [reduce using rule 1 (S)]
    $default    reduce using rule 1 (S)
```

Les crochets encadrant "reduce using rule 1" indique que cette action n'est pas prise en compte par l'analyseur.

Conflit Reduce/Reduce

Dans un conflit Reduce/Reduce yacc choisit d'utiliser la première règle dans l'ordre de description de la grammaire du source yacc. Il est extrèmement périlleux d'utiliser cette caractéristique dans un analyseur car l'ordre des règles de production dans le source yacc peut souvent varier dans la phase de conception du langage.

Conflits multiples

Un autre exemple de gestion des conflits dans yacc consiste à voir les tables obtenues pour la grammaire non LR(1) $G = (\{a, b, c\}, \{S', S, A, B\}, \{S' \to S, S \to Aaa|Bab|aac, A \to a, B \to a\}, S)$.

```
state 1
    S  ->  'a' . 'a' 'c'   (rule 3)
    A  ->  'a' .   (rule 4)
    B  ->  'a' .   (rule 5)

    'a'         shift, and go to state 4
    'a'         [reduce using rule 4 (A)]
    'a'         [reduce using rule 5 (B)]
state 4
    S  ->  'a' 'a' . 'c'   (rule 3)

    'c'         shift, and go to state 7
```

L'action Shift a bien été privilégiée par rapport aux deux reduce possibles. Yacc parvient donc à fournir un analyseur pour nombre de grammaires mais attention, cet analyseur ne reconnait que le mot aac, ce qui n'est pas correct vis à vis de la grammaire (ni aab, ni aaa ne sont reconnus). Pour finir, remarquons que toutes les grammaires LR(1), c'est-à-dire nécessitant un seul jeton de prévision, ne sont pas analysable avec la méthode LALR.

Exercice 6 Soit la grammaire d'expression $G = (\{a, -, /, (,)\}, \{E\}, \{E \to a|(E)|E-E|E/E\}, S)$ dans laquelle a peut être considéré comme un littéral entier.

1. Dessiner la collection canonique de G ;
2. Indiquer les suivants de E.
3. Construire la table d'analyse SLR de G ;
4. Indiquer les conflits obtenus et la manière dont bison les résoud ;
5. Donner les règles de priorité et d'associativité afin d'obtenir un automate à pile correct (division prioritaire par rapport à la soustraction et toutes deux associatives à gauche).
6. Indiquer les modifications de la table.

Chapitre 4

Analyse sémantique

Dans ce chapitre, nous allons étudier une certain nombre de techniques concernant la gestion des tables des symboles, la traduction dirigée par la syntaxe, le contrôle de type, ... Nous supposerons à chaque fois que nous utilisons yacc pour l'analyse syntaxique. Il existe d'autres techniques, notamment liées à l'analyse descendante, mais nous ne les aborderons pas ici et nous conseillons l'ouvrage [1] pour leur étude.

4.1 Table des symboles

4.1.1 Généralités

L'analyseur lexical est le premier à introduire des informations dans la table des symboles. Chaque identificateur, de variable, de paramètre, de constante, de fonction, de procédure, de méthode, d'étiquette de branchement, de classe, doit être enregistré puis, lors de l'analyse syntaxique et sémantique, des informations lui seront attachées. Ces informations concernent généralement le type, la portée, la valeur, ... De plus, dans les langages structurés en blocs ou en fonctions, un même identificateur peut être utilisé dans différentes portées ou dans la même portée pour désigner des objets différents.

Par exemple, en C++, on peut avoir :

```
{int i;         // i dans le bloc englobant est un int
  {char i='a';  // i dans le bloc imbriqué est un char
   i(i);        // i est également une fonction
  }
}
```

Dans les langages structurés en blocs, le plus simple est d'associer une table des symboles à chaque bloc. Lorsque le même identificateur désigne différents objets, il sera nécessaire de construire une entrée différente pour chacun de ces objets afin de pouvoir les renseigner. L'identifiant d'une entrée de la table des symboles deviendra donc l'agrégat du nom de l'objet et de sa catégorie. Un exemple classique en C++ ou en Java concerne les méthodes surchargées. L'identifiant d'une méthode est composé du nom de la méthode et de la liste ordonnée des types des paramètres de cette méthode. Bien entendu, ces actions seront le plus souvent effectuées durant l'analyse syntaxique, car l'analyseur lexical n'a pas les moyens de reconnaître les blocs ou les différentes catégories d'objets.

4.1.2 Implémentation d'une table des symboles

Nous donnons ici, un exemple de table des symboles assez couramment utilisé. Elle est constituée d'un tableau de hachage contenant des listes : `Liste tablehash[MAXHASH];`. Cette technique est souvent appelée "hachage par baquet". Un identificateur, ou nom, permet de calculer un entier qui sera l'indice dans le tableau de hachage. Nous donnons, ci-après, un exemple de fonction calculant cet indice à partir de la chaîne de caractères de l'identifiant.

```
int hash(char *nom){     /* retourne une valeur comprise entre 0 et MAXHASH-1 */
    register int hval;          /* valeur courante de hachage */
    register int pos;           /* position dans le nom */
    hval=pos=0;
    while (nom[pos])            /* tq different de \0 */
        hval=((hval<<1)+nom[pos++]) % MAXHASH; /* calcul de la valeur de hash */
    return hval;
}
```

Il est souhaitable que MAXHASH soit un entier premier. Chaque entrée de la table des symboles est alors un élément, une cellule, de la liste. L'analyse lexicale retournera donc un jeton IDENTIFICATEUR ainsi qu'une valeur sémantique (yylval de lex/yacc) sous forme de chaîne de caractères. Ensuite, l'analyseur syntaxique ajoutera une entrée dans la table des symboles. Cette entrée contiendra au moins :

— une chaîne de caractères représentant le symbole,
— la catégorie syntaxique du symbole (variable, attribut, méthode, ...),
— des propriétés spécifiques à chaque catégorie de symbole.

4.2 Gestion des erreurs

Il peut exister des stratégies de récupérations sur erreurs, notamment à l'aide de yacc. La façon la plus simple de gérer les erreurs à la compilation (compile-time) consiste à générer un message d'erreur puis à quitter le processus de compilation. Par exemple, dans le source yacc, on peut écrire en C++ la fonction yyerror suivante :

```
int yyerror(char *s) {
  cerr<<"Erreur de syntaxe à la ligne : "<<numeroLigne<<endl;
  }
```

La variable numeroLigne est une variable globale définie par le source yacc et mise à jour par le source lex.

De même, lors de l'exécution (run-time), l'appel à une procédure du langage cible ayant la même sémantique est la plus simple façon de gérer les erreurs d'exécution. Voici un exemple en C :

```
void erreurExec(char *msg){
  fprintf(stderr,"ERREUR FATALE : %s\n",msg);
  exit(1);
}
```

4.3 Arbre abstrait

L'arbre syntaxique ou arbre de dérivation d'un mot du langage généré par une grammaire est souvent inutilement complexe. Lorsqu'on génére un arbre lors de l'analyse syntaxique, on préfére une représentation condensée appelée arbre abstrait. Dans celui-ci, les mots-clés sont souvent supprimés et les opérateurs remontés sur le noeud père. De plus, on supprime parfois des symboles de priorité tels que les parenthèses. Par exemple, dans une grammaire G_{ETF}, le mot 2+(2+1)*3 donnera l'arbre abstrait suivant :

```
+
 2
 *
  +
   2
   1
  3
```

La construction de l'arbre abstrait associé au remplissage de la table des symboles est généralement effectuée au cours de l'analyse syntaxique. Chaque noeud de l'arbre abstrait doit contenir :

— la catégorie syntaxique du noeud,
— un lien vers chacun de ses fils,
— un lien vers son père,
— éventuellement des informations complémentaires : entrée de la table des symboles, ...

Une fois l'arbre abstrait construit, le compilateur pourra le parcourir à des fins d'analyse sémantique, d'optimisation, de génération de code.

4.4 Traduction dirigée par la syntaxe

4.4.1 Grammaires attribuées

Théorie

Dans une **grammaire attribuée**, on associe à chaque symbole, terminal et non terminal, de la grammaire, un ensemble d'attributs. Un attribut stocke une information typée. On peut avoir des attributs entiers, chaîne de caractères, ... La notation d'un attribut val associé à un symbole X est X.val. La notation de l'ensemble des attributs associé à un symbole est $X\{val_1, val_2, \ldots, val_k\}$. Un symbole sans attribut sera noté simplement X.

A chaque règle de production, correspond une ou plusieurs règles sémantiques indiquant le mode de calcul de certains des attributs. Bien entendu, le calcul de certains attributs dépend d'autres. Lorsque la règle est récursive, même symbole en partie gauche et droite de production, on indice les occurrences de droite pour les distinguer de l'occurrence de gauche.

Définition 10 *Dans une grammaire attribuée, une règle sémantique associé à une règle de production indique le mode de calcul d'un attribut d'une occurence de symbole présent dans la production. Soit la production* $x_0 \rightarrow x_1 x_2 \dots x_n$*, une règle sémantique s'écrit toujours :* $x_i.val = f(x_{i1}.a_{i1}, x_{i2}.a_{i2}, \dots, x_{ik}.a_{ik})$.

Par exemple, le tableau suivant indique le calcul des attributs de la grammaire G_{ETF}. Soit la grammaire attribuée $G_{ETF} = (\{0,1,\dots,9,+,*,(,)\}, \{E\{val\}, T\{val\}, F\{val\}\}, R, E)$ avec les règles de production R, et les règles sémantiques suivantes calculant des valeurs entières (val) :

Production	Règles sémantiques
$E \rightarrow T$	E.val=T.val
$E \rightarrow E_1 + T$	E.val=E_1.val+T.val
$T \rightarrow F$	T.val=F.val
$T \rightarrow T_1 * F$	T.val=T_1.val*F.val
$F \rightarrow (E)$	F.val=E.val
$F \rightarrow 0$	F.val=0
$F \rightarrow 1$	F.val=1
$F \rightarrow 9$	F.val=9

Grammaires attribuées avec Yacc

Avec yacc, chaque symbole est associé à une **unique** valeur sémantique. Cette valeur est du type `YYSTYPE` qui peut être une union de différents types. Ainsi, l'unique attribut de chaque symbole peut être un pointeur sur une structure C ou une instance de classe C++, donc contenir plusieurs informations typées.

La notation de l'attribut associé à un symbole X dans une production $X \rightarrow \alpha$ est \$\$. La notation de l'attribut associé à une occurrence du symbole X dans une production $Y \rightarrow d_1 d_2 d_3 X d_5 d_6$ est \$4, c'est à dire son indice dans la partie droite. Dans une application de l'exemple précédent, l'analyseur lexical fournit une valeur entière associée à chaque jeton CHIFFRE. On peut également associer des règles d'action aux productions. Par exemple, on pourra afficher la valeur de l'attribut calculé. Pour cela, on augmente la grammaire d'un super axiome S avec les règles :

$S \rightarrow E$ \n	Afficher(E.val)

Voici le source yacc implémentant cet exemple :

```
%{                              /* etf.y  */
#include <stdio.h>              /* printf */
#include <ctype.h>              /* isdigit */
#define YYSTYPE int             /* définition explicite de YYSTYPE comme int */
int yylex(void);void yyerror(char *s);
%}
%token CHIFFRE
%%
liste   :       {/* chaine vide sur fin de fichier Ctrl-D */}
        |       liste ligne
        ;
ligne   :       '\n'            {/* ligne vide : expression vide */}
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       expr '\n'       {printf("Résultat : %d\n",$1);}
        ;
expr    :       terme           {$$ = $1; /* par défaut */}
        |       expr '+' terme  {$$ = $1 + $3;}
        ;
terme   :       fact            {$$ = $1;}
        |       terme '*' fact  {$$ = $1 * $3;}
        ;
fact    :       CHIFFRE         {$$ = $1;}
        |       '(' expr ')'    {$$ = $2;}
        ;
%%
```

```
int yylex(void){
  int c=getchar();while(c==' '||c=='\t')c=getchar(); /* filtrage */
  if (isdigit(c)){
    yylval=c-'0';return CHIFFRE;
  }
  else return c;
}
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(void){yydebug=0; return yyparse();}
```

Remarquons que dans yacc, le type par défaut des attributs est entier mais qu'on peut redéfinir YYSTYPE, soit par un `#define`, soit par un `%union{}`. Si le type d'attribut est unique, alors il n'est pas nécessaire d'indiquer le type des attributs des terminaux et des non terminaux. Sinon, il faut utiliser les définitions yacc `%token<typeDeLUnion> JETON` et `%type<typeDeLUnion> nonterminal`.

Attributs hérités et synthétisés

Définition 11 *Un arbre syntaxique ou abstrait pour lequel on indique sur chaque noeud les valeurs des attributs du symbole, est appelé* ***arbre décoré****.*

Lors de l'analyse syntaxique, on construit très fréquemment un arbre abstrait décoré représentant la structure syntaxique et certains éléments sémantiques du programme.

Définition 12 *Dans une règle sémantique associé à une production, un attribut est synthétisé lorsque il est défini par une fonction des valeurs de ses propres attributs et/ou de ceux de ses fils. Pour une production $x_0 \rightarrow x_1 x_2 \dots x_n$, on a donc : $x_0.val = f(x_{i1}.a_{i1}, x_{i2}.a_{i2}, \dots, x_{ik}.a_{ik})$.*

C'est le cas de tous les attributs de l'exemple précédent. En particulier, les attributs des chiffres sont des fonctions constantes. L'analyse ascendante, par exemple avec yacc, permet facilement de calculer les attributs synthétisés. En particulier, si l'on considère un noeud de l'arbre abstrait comme attribut, la construction de cet arbre abstrait peut être réalisée des feuilles vers la racine. En analyse descendante, le calcul des attributs synthétisés doit se faire lors de la remontée postfixe dans le parcours en profondeur.

Définition 13 *Une grammaire est S-attribuée ssi toutes les règles sémantiques calculent des attributs synthétisés.*

Les grammaires S-attribuées peuvent facilement être implémenthées avec Yacc.

Définition 14 *Dans une règle sémantique associé à une production, un attribut est hérité lorsque il est défini par une fonction des attributs de son père et/ou de ses frères dans l'arbre syntaxique.*

L'évaluation de certains attributs hérités (dépendant du père et des frères de gauche (resp. de droite)) est facile en analyse descendante. Il suffit de les calculer lors du parcours en profondeur. Cela devient plus complexe en analyse ascendante.

Définition 15 *Une grammaire est L-attribuée ssi toutes les règles sémantiques calculent des attributs synthétisés et des attributs hérités ne dépendant que d'attributs de leur père et/ou de leurs frères de gauche (Left).*

En analyse ascendante LR, rappelons que parallèlement à la pile des symboles, une pile des attributs (valeurs sémantiques) existe. De plus, rappelons que le symbole non terminal de gauche n'est réduit qu'après que tous ses fils aient été reconnus. Par conséquent, il n'est pas possible d'hériter directement de son père. Par contre, tous les frères gauches du symbole dont l'attribut doit être calculé sont sur la pile au moment de la réduction. On peut donc calculer facilement les attributs ne dépendant que des attributs de frères gauches. Par exemple, une déclaration simple d'un identificateur entier donne lieu aux règles suivantes.

Production	Règles sémantiques	Commentaire
D→INT ID ;	INT.s="entier", ID.h=INT.s	h est hérité, s synth

Pour un attribut hérité du père, l'astuce consiste à aller chercher dans la pile l'attribut d'un "oncle" de gauche. Un exemple classique concerne l'attribution d'un type à une liste d'identificateurs dans une déclaration, comme par exemple en C : `int i,j,k;`.

Soit $G_{type} = (\{INT, CHAR, ID\{h\}, ',', ';'\}, \{D, L\{h\}, T\{s\}\}, R, D)$. Chaque attribut est une chaine de caractères indiquant un type de données `entier` ou `caractère`. Cet attribut est nommé s et est synthétisé pour T, tandis qu'il est nommé h et est hérité pour L et ID. On a les règles de production R, et les règles sémantiques suivantes :

Production	Règles sémantiques	Commentaire
D→T L	L.h=T.s	h est hérité, s synth
T→INT	T.s="entier"	s est une chaîne
T→CHAR	T.s="caractère"	s est une chaîne
L→ID	ID.h=L.h	hérite du père
L→L_1 , ID	ID.h=L.h, L_1.h=L.H	héritent du père

Le premier héritage (L.h=T.s) concerne un frère gauche et peut donc être réalisé en yacc. Par contre, les trois dernières règles sémantiques d'héritage du père (ID.h=L.h, ID.h=L.h, L_1.h=L.H) ne peuvent être mises en oeuvre avec yacc. Aussi, il convient d'imaginer le contenu de la pile au moment où une production de L est en cours de reconnaissance. On a forcément le symbole T avec son attribut T.s, dans l'élément de pile situé sous le premier ID à être réduit (L→ID). Par conséquent, l'attribut d'ID peut être affecté de $pileAttribut[sommet - 1]$, c'est-à-dire de l'attribut de son oncle T. Par la suite, les réductions par $L \rightarrow L_1, ID$ pourront de la même façon affecter à l'attribut d'ID, la valeur de $pileAttribut[sommet - 3]$. Nous avons donc remplacé les règles sémantiques x=L.h par x=T.s. On n'hérite donc plus de son père mais du frère gauche de son père. Cette transformation est possible, avec yacc, en accédant à l'élément de pile correspondant à T et qui est symbolisé par $0. Attention, cette méthode ne peut toutefois pas être généralisé à tous les héritages de père. Il faut étudier soigneusement les différents états que peut prendre la pile au moment de l'exécution de la règle.

Une implémentation yacc de la grammaire précédente de déclarations est donnée ci-après.

L'analyseur lexical

```
%{                          /* declar.l */
#define YYSTYPE char *      /* définition de YYSTYPE car pas dans y.tab.h ! */
#include "y.tab.h"          /* JETONS crees par yacc et definition de yylval */
%}
lettre          ([a-zA-Z])
chiffre         ([0-9])
%%
[ \t]+          {/* filtrer les blancs */}
int             {return INT;}
char            {return CHAR;}
{lettre}({lettre}|{chiffre})*     {yylval=yytext;return ID;}
.|\n            {return yytext[0]; /* indispensable ! */}
%%
int yywrap(){return 1;}         /* pas de continuation sur un autre fichier */
```

L'analyseur syntaxique

```
%{                              /* declar.y */
#include <stdio.h>
#include <string.h>
#define YYSTYPE char *          /* définition de YYSTYPE comme chaine */
int yylex(void);void yyerror(char *s);
int nb; char affich[1024];
%}
%token INT CHAR ID              /* definition des jetons (tous chaines) */
%%
inter   :       {/* chaine vide sur fin de fichier Ctrl-D */}
        |       inter {affich[0]='\0';} ligne
        ;
ligne   :       '\n'            {/* ligne vide : expression vide */}
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       declar '\n'     {printf("%i déclaration(s) : %s\n",nb,affich);
                                 affich[0]='\0';
        }
        ;
declar  :       type liste
        ;
type    :       INT             {$$="entier";}
        |       CHAR            {$$="caractère";}
        ;
liste   :       ID              {
          nb=1;char couple[128];
```

```
          sprintf(couple,"(%s,%s) ",$1,$0); /* héritage */
          strcat(affich,couple);
        }
        |       liste ',' ID    {
          nb++;char couple[128];
          sprintf(couple,"(%s,%s) ",$3,$0); /* héritage */
          strcat(affich,couple);
        }
        ;
%%
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(){yydebug=0;return yyparse();}
```

L'exécution de l'exécutable obtenu donne :

```
int i, j2, k,l
4 déclaration(s) : (i,entier) (j2,entier) (k,entier) (l,entier)
char c
1 déclaration(s) : (c,caractère)
```

4.4.2 Méthode de transformation des grammaires L-attribuées

La méthode précédente, simple et pratique, ne fonctionne pas toujours. Par exemple, soit les productions suivantes :

Production	Règles sémantiques	Commentaire
$S \to aAC$	C.h=A.s	h est hérité, s synth
$S \to bABC$	C.h=A.s	h est hérité, s synth
$C \to c$	C.s=f(C.h)	calcul sur h

Au moment de réduire par $C \to c$, le calcul de C.s nécessite l'accès à C.h c'est-à-dire A.s. Malheureusement, il est impossible de savoir si cet attribut A.s se situe en $pileAttribut[sommet-1]$ ou en $pileAttribut[sommet-2]$!

Par conséquent, une méthode générique de traitement des attributs hérités consiste à faire précéder chaque symbole ayant un attribut hérité par un non terminal "marqueur" dans chaque production. Ces marqueurs ont une seule ε-production et ne sont présents que pour servir d'emplacement dans la pile d'attributs pour contenir les attributs hérités. Cette méthode appliquée aux productions précédentes donne :

Production	Règles sémantiques	Commentaire
$S \to aAM_1C$	$C.h = M_1.s, M_1.h = A.s$	h est hérité, s synth
$M_1 \to \varepsilon$	$M_1.s = M_1.h$	recopie
$S \to bABM_2C$	$C.h = M_2.s, M_2.h = A.s$	h est hérité, s synth
$M_2 \to \varepsilon$	$M_2.s = M_2.h$	recopie
$C \to c$	$C.s = f(C.h)$	calcul sur h

Ainsi, lorsque la réduction par $C \to c$ a lieu, il suffit de regarder en $pileAttribut[sommet-1]$ pour atteindre C.h, c'est-à-dire $M_1.s$ ou bien $M_2.s$. Attention, le calcul des $M_i.h$ est bien entendu adapté : $M_1.h = A.s$ devient $M_1.h = pileAttribut[sommet-1]$ tandis que $M_2.h = A.s$ devient $M_2.h = pileAttribut[sommet-2]$.

Sur le plan théorique, la méthode échoue parfois lorsque l'adjonction des non terminaux marqueurs et de leurs production génère une grammaire non LR. Cela n'arrive que très rarement dans la pratique.

Enfin, dans deux cas, il n'est pas nécessaire d'introduire des marqueurs :

— dans une règle $G \to D_1 \ldots$ avec $D_1.h = G.h$, introduire un marqueur devant D_1 ne sert à rien sauf quand G est l'axiome ;

— dans une règle $G \to D_1 D_2 \ldots D_n$ avec $D_i.h = D_{i-1}.h$, introduire un marqueur devant D_i ne sert à rien.

Exemple 34

Soit une grammaire d'expressions booléennes à évaluation partielle (ou court-circuit). Dans un interpréteur de ces expressions, il n'est pas nécessaire d'évaluer la suite de l'expression lorsque le résultat est déjà connu. Pour réaliser cette évaluation partielle :

— *l'attribut synthétisé* `val` *remontera la* ***valeur*** *calculée (0 pour faux, 1 pour vrai),*

— *tandis que l'attribut hérité* `cal` *sert uniquement à indiquer s'il faut continuer à* ***calculer*** *le résultat de l'expression courante (dans ce cas sa valeur est 1), ou bien s'il est déjà connu (court-circuit et sa valeur est 0). Remarquons qu'en cas de court-circuit, l'analyse syntaxique sera quand même effectuée mais pas l'évaluation.*

Dans un interpréteur, l'unique intérêt de l'évaluation partielle consiste en la possibilité de mettre dans la même expression des conditions causales, par exemple, `if (!feof(f) && fgetchar(f)!='x') ...`

Production	*Règles sémantiques*	*Commentaire*
$S \to E$	*S.val=E.val, E.cal=1*	*au début, il faut calculer*
$E \to 1$	*E.val=1*	*calcul de base*
$E \to 0$	*E.val=0*	*calcul de base*
$E \to E_1 \| \| E_2$	$E_1.cal = E.cal, E_2.cal = (E.cal?!E_1.val : 0)$ $E.val = (E.cal?(E_1.val?1 : E_2.val) : 99)$	*transmission du court-circuit* *calcul de l'expression*
$E \to !E_1$	$E_1.cal = E.cal, E.val = (E.cal?!E_1.val : 98)$	*calcul de l'expression*
$E \to (E_1)$	$E_1.cal = E.cal, E.val = (E.cal?E_1.val : 97)$	*transmission et calcul*

Les valeurs 99, 98 et 97 signalent des valeurs farfelues qui n'ont aucune chance d'être remontées jusqu'à l'axiome : en effet, lorsque `E.cal` *est faux* `E.val` *n'a aucun intérêt car le résultat final est déjà connu !*

La transformation de cette grammaire L-attribuée par l'introduction de marqueurs donne les règles sémantiques suivantes. Remarquons qu'un marqueur M_i précède toujours une expression E dans la pile, ce qui permet d'obtenir facilement l'attribut hérité cal.

Production	*Règles sémantiques*	*Commentaire*
$S \to M_1 E$	$S.val = E.val, M_1.cal = 1; E.cal = M_1.val$	*au début, il faut calculer*
$M_1 \to \varepsilon$	$M_1.val = M_1.cal$	*transmission*
$E \to 1$	*E.val=1*	*calcul de base*
$E \to 0$	*E.val=0*	*calcul de base*
$E \to E_1 \| \| M_2 E_2$	$E_1.cal = E.cal, M_2.cal = (E.cal?!E_1.val : 0), E_2.cal = M_2.val$ $E.val = (E.cal?(E_1.val?1 : E_2.val) : 99)$	*transmission du court-circuit* *calcul de l'expression*
$M_2 \to \varepsilon$	$M_2.val = M_2.cal$	*transmission du court-circuit*
$E \to !M_3 E_1$	$M_3.cal = E.cal, E_1.cal = M_3.val, E.val = (E.cal?!E_1.val : 98)$	*calcul de l'expression*
$M_3 \to \varepsilon$	$M_3.val = M_3.cal$	*transmission du court-circuit*
$E \to (M_4 E_1)$	$M_4.cal = E.cal, E_1.cal = M_4.val, E.val = (E.cal?E_1.val : 97)$	*transmission*
$M_4 \to \varepsilon$	$M_4.val = M_4.cal$	*transmission du court-circuit*

Remarquons que nous avons introduit les marqueurs M_i afin que l'héritage provienne toujours d'un frère gauche ou d'un oncle gauche. Chacun des marqueurs n'utilise en fait qu'un seul attribut puisqu'il recopie toujours $M_i.cal$ dans $M_i.val$. De plus, l'attribut E.cal provient toujours d'un $M_i.cal$. Aussi, plutôt que d'utiliser les notations théoriques un peu lourdes, on utilise une syntaxe à la yacc avec des $i pour représenter les attributs sur la pile.

Production	*Règles sémantiques*	*Commentaire*
$S \to M_1 E$	*$$=$2*	*résultat final*
$M_1 \to \varepsilon$	*$$=1*	*initialisation*
$E \to 1$	*$$=1*	*calcul*
$E \to 0$	*$$=0*	*calcul*
$E \to E_1 \| \| M_2 E_2$	$\$\$ = (\$0?(\$1?1 : \$4) : 99)$	*calcul de l'expression*
$M_2 \to \varepsilon$	$\$\$ = (\$ - 2?!\$ - 1 : 0)$	*transmission du court-circuit*
$E \to !M_3 E_1$	$\$\$ = (\$0?!\$3 : 98)$	*calcul de l'expression*
$M_3 \to \varepsilon$	$\$\$ = \$ - 1$	*on recopie le marqueur précédent*
$E \to (M_4 E_1)$	$\$\$ = (\$2?\$3 : 97)$	*transmission*
$M_4 \to \varepsilon$	$\$\$ = \$ - 1$	*on recopie le marqueur précédent*

Ce qui donne en yacc :

```
        /* evalcc.y  */
%{
          int yylex(void);
          void yyerror(char *s);
          %}
        /* définition de YYSTYPE comme int par défaut */
        /* définition des précédences */
%left '|'
%right '!'
%%
liste   :       /* chaine vide sur fin de fichier Ctrl-D */
        |       liste ligne
        ;
ligne   :       '\n'            /* ligne vide : expression vide */
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       m1 exp '\n'     {printf("Résultat : %d\n",$2);}
```

```
        ;
m1      :                       {$$=1;          /* $$=vrai */}
        ;
exp     :       exp '|' m2 exp  {$$=($0?($1?2:$4):99); /* un peu condensé ! */}
        |       '!' m3 exp      {$$=($0?!$3:98); /* $0 est l'attribut de mi */}
        |       '(' m4 exp ')'  {$$=($2?$3:97);}
        |       '1'             {$$=1;          /* $$=vrai */}
        |       '0'             {$$=0;          /* $$=faux */}
        ;
m2      :                       {$$=($-2?!$-1:0);}
        ;
m3      :                       {$$=$-1;}
        ;
m4      :                       {$$=$-1;}
        ;
%%
int yylex(void) {int c; while(((c=getchar())==' ') || (c=='\t')); return (c);}
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(void){/*yydebug=1*/; return yyparse();}
```

Dans cet évalaluateur à court-circuit, nous avons donné la valeur 2 lorsqu'un court-circuit était réalisé grâce au ou logique. Voici quelques exécutions :

```
0|0|1|0
Résultat : 2
0|0|0|0|1
Résultat : 1
(!!1)
Résultat : 1
!(1|0)|0
Résultat : 0
```

Exercice 7 Compléter l'évaluateur booléen en ajoutant la règle du et logique à court circuit. Compléter le source yacc.

Chapitre 5

Génération de code

5.1 Introduction

On utilise généralement un langage intermédiaire entre le langage évolué et le langage de la machine hôte.

— Deux frontaux ("front-end") de gcc et g++, qui traduisent le fichier source en une représentation interne arborescente commune : Register Transfer Language (RTL). Inspiré de Lisp ce langage a une représentation interne, structures chainées par pointeurs, et textuelle aux fins de débogage. Pour lire cette apparence textuelle : `gcc -dr exrtl.c; cat exrtl.c.rtl`. Cette représentation dépend tout de même de la machine cible et n'est donc pas totalement portable. La seconde partie finale ("back-end", bulk compiler), est commune à gcc et g++ pour une machine donnée.

— Le byte-code de Java est un langage universel qu'interprète une machine virtuelle. La portabilité des .class est donc totale à condition d'avoir un interprèteur (java, machine virtuelle) sur la machine cible. Le langage byte-code est assez proche d'un langage machine, à ceci près qu'il utilise beaucoup la pile et des variables locales plutôt que des registres. Il contient environ 200 instructions, ce qui permet de stocker le code opération sur un octet.

— Le P-code du Pascal est l'un des premiers langages intermédiaires à avoir été utilisé par un compilateur. C'est un langage pour machine abstraite à pile (on voit la filiation avec Java).

Le langage intermédiaire est souvent soit un langage de machine virtuelle à pile, soit un langage d'arbre représenté par une notation postfixée. Sans en étudier tous les détails, la section suivante illustre le fonctionnement d'une machine à pile.

5.2 Machine virtuelle à pile

Une machine, virtuelle ou abstraite, à pile est constituée :

— d'une mémoire d'instructions et d'un compteur ordinal CO,

— d'une mémoire de données,

— d'une pile.

Les instructions de la mémoire d'instructions sont exécutées en séquence. Les différentes instructions sont rangées en catégories :

— manipulation de la pile : empiler, dépiler des constantes ou des données de la mémoire, opérer sur le ou les 2 sommets de pile et le ou les remplacer par le résultat.

— contrôle du flux d'instructions : branchements conditionnels, appels et retours de procédure.

L'utilisation de la pile est continuelle puisque les opérandes sont stockés dessus pour les opérations arithmétiques, logiques, de branchements ou d'appels. Pour plus d'informations sur ce type de langage, voir par exemple l'ouvrage [4].

5.3 Développement d'un compilateur

Dans le cadre d'un projet de développement d'un compilateur, l'étude du langage source est fondamentale mais n'est pas suffisante. En effet, le choix d'un "bon" langage intermédiaire et du langage de développement du compilateur est important. Tout d'abord, de nos jours, il est impensable d'écrire un compilateur en langage d'assemblage. Dans l'environnement Unix, l'écriture en C permet d'obtenir une excellente efficacité (le système est lui-même majoritairement écrit en C). L'utilisation d'un langage intermédiaire facilite l'écriture de la partie finale du compilateur pour différentes machines. Dans la famille de compilateurs gnu (gcc, ...), on peut spécifier la correspondance des instructions RTL et de la machine cible dans un fichier, ce qui permettra de générer du code machine sans réécrire cette partie finale !

Un compilateur peut être représenté par une forme géométrique en T, notée S_IO, où S est le langage source, O le langage objet, et I le langage d'implémentation du compilateur. Par exemple, un compilateur écrit en C++ traduisant du Pascal en C est noté : $Pascal_{C++}C$. Ces formes en T peuvent être imbriquées, représentant en ceci la composition de compilateurs. Ainsi, si nous disposons d'un second compilateur C++ en langage machine, la compilation de $Pascal_{C++}C$ par $C++_MM$ fournit un compilateur de Pascal en C écrit en langage machine.

Cette technique de compilation de compilateur a souvent été utilisée dans la technique d'auto-amorçage. Pour un langage L dont on souhaite écrire un compilateur pour la machine M, cette technique consiste à écrire un premier compilateur grossier en L L'_LM, puis à traduire à la main ce compilateur dans le langage M, on obtient donc L'_MM. Ensuite, on utilise ce premier compilateur grossier pour recompiler le compilateur écrit en L : ce compilateur s'est compilé lui-même ! De la même façon, le premier interpréteur Lisp a été écrit en Lisp puis traduit à la main. De nouvelles modifications du compilateur sont ensuite utilisées pour l'affiner.

Les techniques de compilation de compilateur sont également utilisées pour les compilateurs croisés. Supposons que l'on a écrit un compilateur L en L générant du code pour la machine N : L_LN. Si l'on a à sa disposition un compilateur de L sur une autre machine M, L_MM, alors on peut très bien obtenir une version du compilateur fonctionnant sur la machine N de la façon suivante :

1. compiler L_LN gâce à L_MM : on obtient L_MN qui est un compilateur.
2. compiler encore une fois L_LN gâce à ce nouveau compilateur L_MN : on obtient donc L_NN.

Remarquons que l'on a conçu un compilateur tournant sur la machine N, sans jamais utiliser la machine N. Il suffit de connaître les spécifications de cette machine avant même qu'elle ne soit construite.

Pour ces deux raisons, auto-amorçage et compilation croisée, mais aussi afin de tester la puissance du langage en cours de développement, il est souvent intéressant d'écrire un compilateur dans son propre langage source.

Chapitre 6

Sémantique opérationnelle des langages de programmation

6.1 Introduction

Ce chapitre étudie différents modèles de programmation et leur implantation sur les machines informatiques classiques (modèle de Von Neumann). Les langages utilisés pour illustrer nos propos seront le C, le C++, Java. Le C est un langage évolué qui est en même temps très proche de la machine ; il est donc couramment utilisé pour écrire des compilateurs, des systèmes d'exploitation, ... Les langages à objets C++ et Java introduisent un niveau conceptuel supplémentaire dans la programmation. Il est cependant utile de connaître leur implantation afin de programmer presque aussi efficacement avec ces langages qu'avec le C.

6.2 Organisation de l'espace mémoire

6.2.1 Image mémoire

Après compilation et édition de liens d'un ensemble de sources, le fichier exécutable est **chargé** en mémoire centrale pour exécution. Le chargeur, partie indispensable du système d'exploitation, va donc installer les différentes parties de l'exécutable dans des blocs de mémoire que nous appellerons **segments**. Ces segments auront été réservés par le chargeur auprès du système de gestion de mémoire. Il y a au moins quatre segments pour des langages tels que C, C++, Pascal, ... :

- le segment de code contient les instructions machines à exécuter ;
- le segment de données **statiques** contient les variables globales et/ou statiques. Ces variables sont créées à la compilation ("compile time") d'où leur qualificatif de statique ;
- le segment de pile est vide au début et contiendra les adresses de retour, les paramètres, les variables locales de chaque procédure ;
- le segment de tas ou de données **dynamiques** est vide au début et contiendra les objets créés dynamiquement (malloc, new, ...). Ces variables ou objets sont créées à l'exécution ("run time") d'où leur qualificatif de dynamique. Remarquons que les objets de la pile devraient également être qualifiés de dynamique !

L'ensemble de ces segments est appelée image mémoire du processus. Une fois l'image mémoire installée par le chargeur, le processeur peut commencer à exécuter le segment de code de ce nouveau processus. A la mort de ce dernier, il ne restera qu'à désallouer les segments mémoires maintenant inutiles.

6.2.2 Appel procédural

La notion de fonction n'existe pas au niveau machine, seule les procédures, sans paramètre résultat, permettent d'implanter les fonctions. Le ou les paramètres résultats (out) sont installés sur la pile par l'appelant juste avant les paramètres de données (in), puis l'appel a lieu (CALL). L'appelé, sauve le registre de base de pile de l'appelant sur la pile, positionne son registre de base au sommet de pile courant, puis installe ses propres variables locales. La pile est également utilisée pour stocker les objets **temporaires**, résultats d'expressions en cours d'évaluation. L'ensemble de ces informations résidant sur la pile et constituant le contexte d'une instance de procédure est souvent appelé enregistrement d'activation, ou bloc d'activation, ou bloc de pile. En anglais, le terme consacré est "stack frame", et il est très utile en débogage (up/down).

L'ordre sur la pile des paramètres de données varie selon les langages de programmation. Il faut noter qu'en C et C++, l'existence de fonctions à nombre d'arguments variables (tel printf) impose d'empiler les arguments à l'envers par rapport à leur énumération dans la signature de la fonction.

6.2.3 Passage des paramètres

Les différents types de passages de paramètres sont une caractéristique des langages de programmation :

— en C, seul le **passage par valeur** existe : le paramètre effectif est recopié sur la pile. L'appelé utilise ensuite cette copie en lecture/écriture sans risque pour le paramètre effectif. Le passage de pointeur permet à l'appelé de modifier l'objet pointé, mais constitue tout de même un passage par valeur (recopie du pointeur) ;

— en C++, on peut également utiliser le **passage par référence**. C'est l'adresse de l'objet qui est mis sur la pile et non sa valeur. L'appelé utilise ensuite cet objet en lecture/écriture en accédant directement à lui. De plus, la possibilité de qualifier la référence de constante (const) permet d'interdire à l'appelé de modifier l'objet. Ceci est utile pour les gros objets.

— en Java, les deux types précédents existent mais sont implicites en fonction du type du paramètre : si le paramètre est de type primitif (int, char, ...), le passage a lieu par valeur, sinon, le paramètre est une référence à un objet (String, Vector, ...), le passage a lieu par référence. A noter que la qualification constante n'existe pas.

Il existe d'autre façon de passer les paramètres sur la pile, notamment par copie et recopie à l'appel et au retour. En Ada, chaque paramètre appartient à une catégorie "in", "out" ou "inout". Cette spécification de haut niveau cache la manière de passer les paramètres, ce qui est excellent pour le programmeur qui n'a plus à se soucier de ces problèmes techniques. En Java, les objets sont tous dynamiques, c'est-à-dire créés à l'exécution, et sont toujours passés par référence, donc modifiables.

6.2.4 Accès aux noms (liaison)

Dans cette section, le terme "nom" désigne aussi bien une donnée, c'est-à-dire une portion de mémoire (variable, tableau, objet, ...), qu'une procédure.

Accès aux données locales

Les objets locaux, paramètres ou variables locales, sont accédés via le registre pointeur de base de pile indexé par un déplacement. Le calcul de ce déplacement est effectué à la compilation.

Accès aux données dynamiques

Les objets dynamiques sont accédés via un pointeur (C, C++) ou une référence (C++, Java). Ce pointeur est lui même un objet local ou dynamique ou ... La valeur de ce pointeur est calculée à l'exécution.

Accès aux noms statiques

Les noms statiques, variables globales ou de classe ou statiques fichier ou statiques fonction pour les données, fonctions globales ou statiques ou méthodes de classes ou méthodes d'instance non virtuelles (C++), sont accédés via une adresse calculée à la compilation. Cette adresse est le plus souvent un déplacement par rapport au début du segment de données ou du segment de code.

Accès aux noms non locaux

La notion de bloc {} permet de préciser la portée des noms définis à l'intérieur d'un bloc. En cas de blocs imbiqués, l'accès à un nom est réalisé par recherche de l'objet depuis le bloc courant puis en remontant dans les blocs englobants. Cette règle peut être implantée en associant à chaque bloc un bloc de pile associé (frame). Un bloc peut alors être vu comme une procédure sans paramètres. Une autre façon de faire consiste à ne constituer qu'un seul bloc de pile pour tous les blocs imbriqués. Dans ce dernier cas, la résolution est forcément statique. Dans le premier cas, la recherche de l'objet nommé peut être effectuée dynamiquement en parcourant les blocs de piles. Remarquons, qu'en Pascal, les procédures comme les données peuvent être imbriquées.

6.3 Langages à objets

Nous étudierons quelques caractéristiques des langages C++ et Java. Ces deux langages à objets sont des langages à classe. Les objets sont donc des instances d'une classe qui a été définie à la compilation : les classes donc la taille et la structure des instances sont connues à la compilation.

6.3.1 C++

En C++, les objets peuvent être globaux, locaux (automatiques), dynamiques (new).

Données membres

Les données membres statiques (attributs de classe) sont stockées en une seule occurrence dans le segment de données statique. La liaison est donc effectuée à la compilation. On peut les assimiler à des variables globales, sauf en ce qui concerne le contrôle d'accès (public, private) et la portée. L'initialisation de toutes les données du segment de données statique les concerne.

Les données membres non statiques (attributs d'instance) sont stockées dans chaque instance. La liaison est effectuée à la compilation pour la partie déplacement à l'intérieur de la "struct". Des contraintes d'**alignement** nécessitent souvent une taille d'instance supérieure à la somme des tailles des attributs d'instance. Les classes vides d'attribut ont une taille par défaut de 1.

Fonctions membres

Les fonctions membres non virtuelles sont stockées dans le segment de code. La liaison est effectuée à la compilation. Le coût à l'exécution est donc identique à celui de fonctions externes (à la C).

Les données membres **virtuelles** sont stockées dans un tableau de fonctions virtuelles propre à la classe. Chaque instance contient un pointeur (vptr) sur cette table de pointeurs de fonctions. La liaison est effectuée à l'exécution au prix d'une indirection. Remarquons que la programmation par objets préconise la "virtualisation" des fonctions en raison justement de cette liaison tardive.

Exemple 35

```
class Vide{};
class UnIntUnChar{public:int i; private:char c;};
class UneFonction{int f(){return 0;}};
class UneVirtuelle{virtual int f(){return 0;}};

main(){
  cout<<"Taille de Vide : "<<sizeof(Vide)<<endl;
  cout<<"Taille de UnIntUnChar : "<<sizeof(UnIntUnChar)<<endl;
  cout<<"Taille de UneFonction : "<<sizeof(UneFonction)<<endl;
  cout<<"Taille de UneVirtuelle : "<<sizeof(UneVirtuelle)<<endl;
}
```

Une exécution donne le résultat suivant :

```
Taille de Vide : 1
Taille de UnIntUnChar : 8
Taille de UneFonction : 1
Taille de UneVirtuelle : 4
```

Héritage simple

Le C++ permet l'héritage multiple. Dans l'héritage simple, une classe dérive d'une classe de base. Pour l'implantation d'une instance dérivée, la règle fondamentale est le respect de l'intégrité du sous-objet. Les attributs de base et dérivés ne sont pas "compactés".

Exemple 36

```
class UnInt2Char:UnIntUnChar{public:char c2;};
cout<<"Taille de UnInt2Char : "<<sizeof(UnInt2Char)<<endl;
```

Une exécution donne le résultat suivant :

```
Taille de UnInt2Char : 12
```

Héritage multiple

La règle d'intégrité du sous-objet est appliquée pour chaque sous-objet hérité. La taille de l'instance dérivée est égale à la somme des tailles des classes de base.

Exemple 37

```
class Multiple:UnIntUnChar, Vide{};
cout<<"Taille de Multiple : "<<sizeof(Multiple)<<endl;
```

Une exécution donne le résultat suivant :

```
Taille de Multiple : 12
```

Ici la taille de (8+1) a été augmentée à 12 pour des raisons d'alignement.

Héritage de classes polymorphes

Les classes polymorphes sont celles qui contiennent au moins une fonction virtuelle. Dans ce cas, un pointeur de table de fonctions virtuelles (vptr) est placé dans chaque instance (au début ou à la fin selon les compilateurs). Une classe héritant de plusieurs classes polymorphes contiendra donc chaque sous-objet, y compris un "vptr" pour chacun. Chaque fonction virtuelle est affectée à un **indice fixe** dans la table des fonctions virtuelles. Ainsi, dans le cas d'une redéfinition d'une méthode de base, la nouvelle fonction est installée à la place de celle de base. Ceci permet de déterminer à la compilation la fonction à exécuter.

Exemple 38

```
class UneVirtuelleBis{virtual int g(){return 0;}};
class DeuxVirtuelleBis:UneVirtuelle, UneVirtuelleBis{};
cout<<"Taille de DeuxVirtuelleBis : "<<sizeof(DeuxVirtuelleBis)<<endl;
```

Une exécution donne le résultat suivant :

```
Taille de DeuxVirtuelleBis : 8
```

Ici la taille de (4+4) est celle des deux "vptr".

6.3.2 Java

En Java, les objets sont exclusivement dynamiques (new). Toutes les méthodes sont virtuelles, ce qui permet une programmation fortement polymorphe. Une classe de base "Object" est la racine de la hiérarchie d'héritage. L'héritage multiple n'est pas permis. Par contre, une classe peut implémenter plusieurs interfaces, signature publique d'une classe. La compilation d'un fichier source .java génère un fichier de "byte-code" .class. Un interpréteur, ou machine virtuelle java ("Java Virtual Machine"), exécute ensuite le fichier .class. L'intérêt de cette architecture réside dans la portabilité totale des .class sous différents environnements (Unix, Windows, MacOS, ...). De plus, l'ensemble des navigateurs internet (netscape, explorer, ...) possèdent une JVM intégrée, ce qui garantit l'exécutabilité sur la majorité des machines.

Index

AFD, 5
Analyse descendante, 19, 21
analyse lexicale, 5
analyse syntaxique, 19
arbre abstrait, 19, 50
associativité, 35
attribut, 33
automate, 5
automate à pile, 21

bison, 31

catégorie lexicale, 7
conflit, 29, 46

expression, 3

factorisation, 26
filtrage, 7
flex, 10
frontal, 57

gestion des erreurs, 50
grammaire attribuée, 50

identificateur, 3
instruction, 3

jeton, 7

LALR(1), 31
langage intermédiaire, 57
lex, 10
lexème, 7
littéral, 3
LL(1), 30
LR, 42

mot-clé, 3

opérateur, 4

premiers, 26
priorité, 35

récursivité à gauche, 24

SLR, 43
suivants, 27

table des symboles, 49
token, 7

yacc, 31

Bibliographie

[1] Ravi Sethi, Jeffrey David Ullman, and Alfred Vaino Aho. *Compilateurs : Principes, techniques et outils*. InterEditions, 1990. La bible, indispensable pour la compilation, aspect info., 870 pages.

[2] Arto Salomaa. *Formal languages*. ACM monograph series. Academic Press, New York, NY, 1973. Très formel, en anglais, aspect maths, 320 pages.

[3] David Flanagan. *Java in a Nutshell*. " O'Reilly Media, Inc.", 2005. Le livre de référence sur java, 600 pages.

[4] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. Le byte-code de java, 420 pages.

[5] S. B. Lippman. *Le modèle objets du C++*. Int. Thomson Pub., 1996. Très technique, peu pédagogique, les entrailles du C++, 260 pages.

[6] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. L'ouvrage de référence sur le C++ et son implémentation, 440 pages.

[7] John R Levine, Tony Mason, and Doug Brown. *Lex & yacc*. " O'Reilly Media, Inc.", 1992. Pour apprendre à programmer en lex et yacc, 330 pages.

Solutions des exercices

Solution 2 — Grammaire non ambiguë et non récursive à gauche : $S \to aSbS|\varepsilon$
— Programme C :

```
/**@file dyck.c
 *@author Michel Meynard
 *@brief Analyse descendante récursive de mots de Dyck
 */
#include <stdio.h>
#include <stdlib.h>

#define AVANCER {jeton=getchar();numcar++;}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu : erreur de syntaxe \
au caractère numéro %d \n",numcar); exit(1);}
int jeton;                        /* caractère courant du flot d'entrée */
int numcar=0;                     /* numero du caractère courant (jeton) */

void S(void){ /* AXIOME */
  if (jeton=='a') {             /* regle : S->aSbS */
    AVANCER
    S();
    TEST_AVANCE('b')
    S();
  }
  else ;                        /* regle : S->epsilon */
}
int main(void){ /* Fonction principale */
  AVANCER                       /* initialiser jeton sur le premier car */
  S();                          /* axiome */
  if (jeton==EOF)               /* expression reconnue et rien après */
    printf("\nMot reconnu\n");
  else ERREUR_SYNTAXE           /* expression reconnue mais il reste des car */
  return 0;
}
```

Solution 4
```
%{#include  <stdio.h>
  int yylex(void);  void yyerror(char *s);
%}
%%
S : S 'a' S 'b'  {}
| {}
%%
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int yylex(){return getchar();}
int main(void){
  yydebug=0;
  if (!yyparse()) /* appel à l'analyseur généré par yacc */
    printf("\nMot de Dyck reconnu\n");
  else
    printf("\nMot non reconnu\n");
  return 0;
```

}

Solution 5 AFD : $I_0 = Fermeture(\{S' \to .S\}) = \{S' \to .S, S \to .aSbS, S \to \varepsilon.\}$
$T = \{(I_0, a, I_1)\}$
$I_1 = Fermeture(\{S \to a.SbS\}) = \{S \to a.SbS, S \to .aSbS, S \to \varepsilon.\}$
$T+ = \{(I_1, a, I_1)\}$
$T+ = \{(I_1, S, I_2)\}$
$I_2 = Fermeture(\{S \to aS.bS\}) = \{S \to aS.bS\}$
$T+ = \{(I_2, b, I_3)\}$
$I_3 = Fermeture(\{S \to aSb.S\}) = \{S \to aSb.S, S \to .aSbS, S \to \varepsilon.\}$
$T+ = \{(I_3, a, I_1)\}$
$T+ = \{(I_3, S, I_4)\}$
$I_4 = Fermeture(\{S \to aSbS.\}) = \{S \to aSbS.\}$
$T+ = \{(I_0, S, I_5)\}$
$I_5 = Fermeture(\{S' \to S.\}) = \{S \to S.\}$

Table d'analyse :

	Action			Successeur
	a	b	$	S
0	S1	R($S \to \varepsilon$)	R($S \to \varepsilon$)	5
1	S1	R($S \to \varepsilon$)	R($S \to \varepsilon$)	2
2		S3		
3	S1	R($S \to \varepsilon$)	R($S \to \varepsilon$)	4
4		R($S \to aSbS$)	R($S \to aSbS$)	
5			Accepter	

Avec le mot *abaababb*$, empilement de :
aSbaaSbaSbS avant la première réduction intéressante (R($S \to aSbS$))

Solution 6 1. collection canonique : en ajoutant le super axiome S et la règle $S \to E$.

$I_0 = Fermeture(\{S \to .E\}) = \{S \to .E, E \to .a, E \to .(E), E \to .E - E, E \to .E/E\}$
$T = \{(I_0, E, I_1), (I_0, a, I_2), (I_0, (, I_3), \}$
$I_1 = Fermeture(\{S \to E., E \to E. - E, E \to E./E\}) = \{S \to E., E \to E. - E, E \to E./E\}$ $T+ = \{(I_1, -, I_4), (I_1, /, I_5)\}$
$I_2 = Fermeture(\{E \to a.\}) = \{E \to a.\}$ $I_3 = Fermeture(\{E \to (.E)\}) = \{E \to (.E), E \to .a, E \to .(E), E \to .E - E, E \to .E/E\}$ $T+ = \{(I_3, E, I_6), (I_3, a, I_2), (I_3, (, I_3)\}$
$I_4 = Fermeture(\{E \to E - .E\}) = \{E \to E - .E, E \to .a, E \to .(E), E \to .E - E, E \to .E/E\}$ $T+ = \{(I_4, E, I_7), (I_4, a, I_2), (I_4, (, I_3)\}$
$I_5 = Fermeture(\{E \to E/.E\}) = \{E \to E/.E, E \to .a, E \to .(E), E \to .E - E, E \to .E/E\}$ $T+ = \{(I_5, E, I_8), (I_5, a, I_2), (I_5, (, I_3)\}$
$I_6 = Fermeture(\{E \to (E.), E \to E. - E, E \to E./E\}) = \{E \to (E.), E \to E. - E, E \to E./E\}$ $T+ = \{(I_6,), I_9), (I_6, -, I_4), (I_6, /, I_5)\}$
$I_7 = Fermeture(\{E \to E - E., E \to E. - E, E \to E./E\}) = \{E \to E - E., E \to E. - E, E \to E./E\}$ $T+ = \{(I_7, -, I_4), (I_7, /, I_5)\}$
$I_8 = Fermeture(\{E \to E/E., E \to E. - E, E \to E./E\}) = \{E \to E/E., E \to E. - E, E \to E./E\}$ $T+ = \{(I_8, -, I_4), (I_8, /, I_5)\}$
$I_8 = Fermeture(\{E \to (E).\}) = \{E \to (E).\}$

2. $Suivants(E) = \{-, /,), \$\}$
3. table

	a	-	/	(	)	$	E
0	S2			S3			1
1		S4	S5			ACCEPTER	
2		R($E \to a$)	R($E \to a$)		R($E \to a$)	R($E \to a$)	
3	S2			S3			6
4	S2			S3			7
5	S2			S3			8
6		S4	S5		S9		
7		S4,R($E \to E - E$)	S5,R($E \to E - E$)		R($E \to E - E$)	R($E \to E - E$)	
8		S4,R($E \to E/E$)	S5,R($E \to E/E$)		R($E \to E/E$)	R($E \to E/E$)	
9		R($E \to (E)$)	R($E \to (E)$)		R($E \to (E)$)	R($E \to (E)$)	

4. Il y a donc 4 conflits décalage/réduction. Bison résoud les conflits en privilégiant le décalage. Donc, sémantiquement les opérations seront exécutées de droite à gauche. Par exemple, $1/2 - 6/2 = 1/(2 - (6/2)) = -1$!
5. priorités et associativités

```
%left '-'
%left '/'
```

6. table

	a	-	/	(	)	$	E
7		R($E \to E - E$)	S5		R($E \to E - E$)	R($E \to E - E$)	
8		R($E \to E/E$)	R($E \to E/E$)		R($E \to E/E$)	R($E \to E/E$)	

Solution 7

$E \to E_1 \&\& M_5 E_2$	$\$\$ = (\$0?(\$1?\$4 : 0) : 96)$	calcul de l'expression
$M_5 \to \varepsilon$	$\$\$ = (\$ - 2?\$ - 1 : 0)$	transmission du court-circuit

Ce qui donne en yacc :

```
        /* evalccet.y  */
%{
          int yylex(void);
          void yyerror(char *s);
          %}
        /* définition de YYSTYPE comme int par défaut */
        /* définition des précédences */
%left '|'
%left '&'
%right '!'
%%
liste   :       /* chaine vide sur fin de fichier Ctrl-D */
        |       liste ligne
        ;
ligne   :       '\n'            /* ligne vide : expression vide */
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       m1 exp '\n'     {printf("Résultat : %d\n",$2);}
        ;
m1      :                       {$$=1;          /* $$=vrai */}
        ;
exp     :       exp '|' m2 exp  {$$=($0?($1?2:$4):99); /* un peu condensé ! */}
        |       '!' m3 exp      {$$=($0?!$3:98); /* $0 est l'attribut de mi */}
        |       '(' m4 exp ')'  {$$=($2?$3:97);}
        |       '1'             {$$=1;          /* $$=vrai */}
        |       '0'             {$$=0;          /* $$=faux */}
        |       exp '&' m5 exp  {$$=($0?($1?$4:0):96); /* un peu condensé ! */}
        ;
m5      :                       {$$=($-2?!$-1:0);}
        ;
m2      :                       {$$=($-2?!$-1:0);}
        ;
m3      :                       {$$=$-1;}
        ;
m4      :                       {$$=$-1;}
        ;
%%
int yylex(void) {int c; while(((c=getchar())==' ') || (c=='\t')); return (c);}
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(void){/*yydebug=1*/; return yyparse();}
```

Voici quelques exécutions :

```
0|1&0|1
Résultat : 1
0&1&1|1&0
```

```
Résultat : 0
1&0|1&1|0|1
Résultat : 2
```