Analyse Syntaxique et Interprétation HAI601I

Michel Meynard

UM

Univ. Montpellier

Table des matières I

Plan

Plan

Objectifs I

- 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".

Plan

Familles de grammaires et de langages : hiérarchie de Chomsky I

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 : $r = \alpha X \beta \rightarrow \alpha m \beta$ 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

Familles de grammaires et de langages : hiérarchie de Chomsky II

$S \to \varepsilon$ peut exister à condition que S ne soit pas présente dans une partie droite d'une règle de production.

Exemple

le P garçon $\to$ le petit garçon ; la P N $\to$ la petite N ; N $\to$ 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

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

Familles de grammaires et de langages : hiérarchie de Chomsky III

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

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

Théorème

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

Plan

Langages réguliers : propriétés et caractérisations I

Théorème

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 (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.

Langages réguliers : propriétés et caractérisations II

Théorème (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

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

Plan

Langages algébriques : propriétés et caractérisations I

Définition

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 $X \rightarrow y_1 y_2 \dots y_n$ 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.

Langages algébriques : propriétés et caractérisations II

Théorème

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

Définition

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

Tout langage régulier est non ambigu.

Langages algébriques : propriétés et caractérisations III

Théorème (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

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

Théorème

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

Plan

Types de traducteurs I

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

Plan

Modèle classique de compilation I

1. Analyse du source :
   1. lexicale : découpage en “jetons” (tokens) ;
   2. syntaxique : vérification de la correction grammaticale et production d'une représentation intermédiaire (souvent un arbre) ;
   3. 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 :
   1. génération de code intermédiaire : machine abstraite (ou virtuelle), p-code du Pascal, byte-code de java, basic tokenisé de Visual Basic, . . . ;

Modèle classique de compilation II

2. optimisation de code : optimiseur de requêtes SQL, optimiseurs C et C++, ...;
3. 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).

Modèle classique de compilation III

- 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.

Remarques I

- 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.

Plan

Plan

Reconnaissance d'un mot par un AFD I

rappelons les résultats sur les Automates d'états Finis Déterministes (AFD) :

- un AFD possède un unique état initial
- aucun couple de transitions $(e_i, a, e_j), (e_i, a, e_k)$ tels que $j \neq k$
- l'ensemble des transitions peut être implémenté simplement par une table à double entrée $TRANS[etatCourant][carCourant]$ qui contient l'état suivant
- l'algorithme suivant décrit la reconnaissance d'un mot par un AFD

Algorithme accepter()

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*$(etat \in A)$;

Plan

Implémentation d'Automates Finis Déterministes AFD I

Soit l'AFD de la figure 1 reconnaissant l'expression régulière $a(b^+c)?|bd$

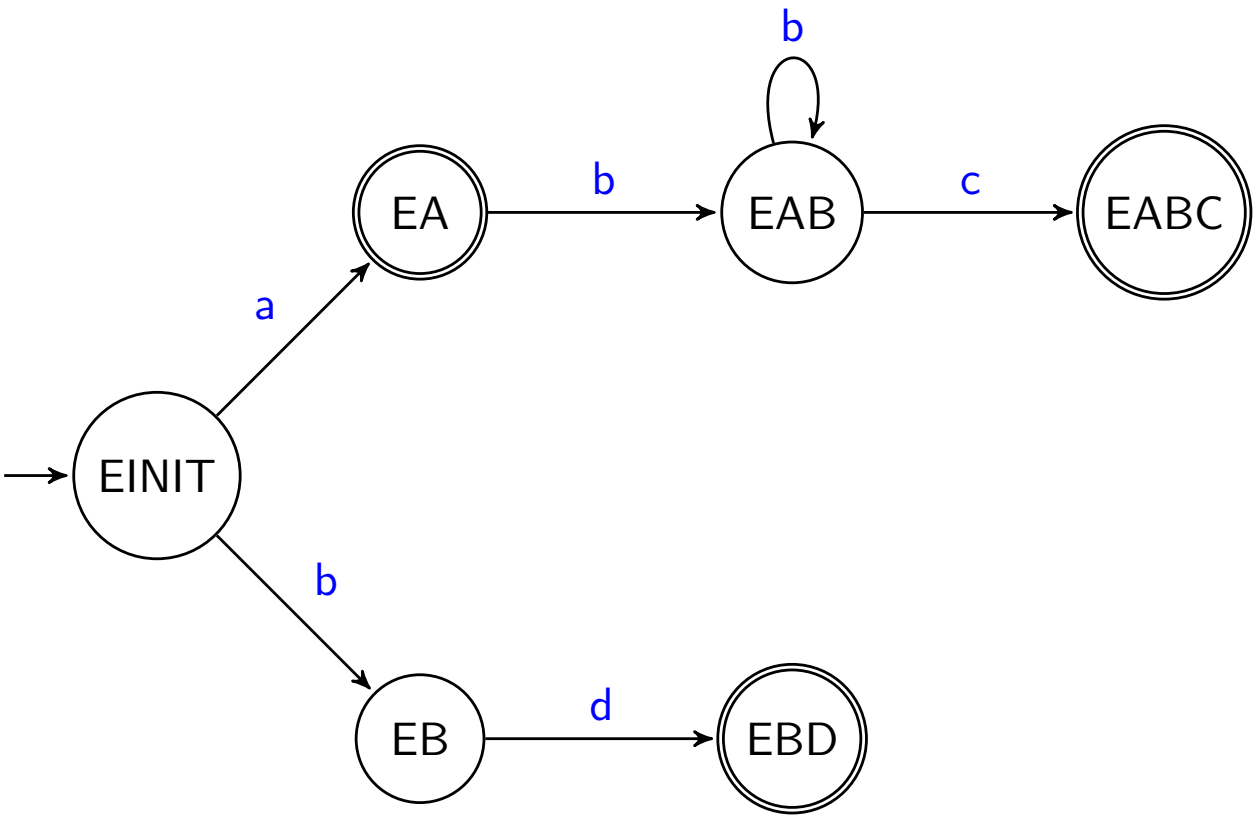


Figure 1 – AFD

Implémentation d'Automates Finis Déterministes AFD II

Nous le représentons par un fichier d'en-tête C ayant les caractéristiques suivantes :

- les états de d'automate sont représentés par des macro définitions symboliques (`#define`)
- le vocabulaire sera défini sur un sous-ensemble (ici $\{a, b, c, d\}$) du type C char
- les transitions sont stockées dans un tableau TRANS d'entiers à double entrée de NBETAT lignes et 256 colonnes (un char possèdant 256 codes)
- un tableau FINAL d'entier de taille NBETAT indiquera si l'état est final (1) ou non (0)

Implémentation d'Automates Finis Déterministes AFD III

```
/**
 * @file afd.h  Définition d'un AFD reconnaissant a(b+c)?|bd
 * @author Michel Meynard
 */
typedef enum { EINIT=0, EA, EAB, EABC, EB, EBD, NBETAT} Etat; //
↪  ATTENTION NBETAT doit être le dernier

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

void creerAfd(){             /* Construction de l'AFD */
  for (int i=0;i<NBETAT;i++){
    for(int 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;
```

Implémentation d'Automates Finis Déterministes AFD IV

```
  TRANS[EAB]['c']=EABC;TRANS[EINIT]['b']=EB;TRANS[EB]['d']=EBD;
  FINAL[EA]=FINAL[EABC]=FINAL[EBD]=1; /* états finaux */
}
```

Implémentation en C de l'algorithme de reconnaissance d'un mot par un AFD I

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

int accepter(){                              /* reconnaît un mot sur
↪  l'entrée standard */
  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 */
```

Implémentation en C de l'algorithme de reconnaissance d'un mot par un AFD II

```
  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;
}
```

Test du programme I

```
$ gcc -Wall accepter.c -o accepter
$ 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 !
```

Discussion sur l'implémentation I

Il existe d'autres types d'implémentation, plus efficaces en mémoire, 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 flex (voir section 5), avec une structure de données réduisant la taille de la table qui est souvent "creuse"

Plan

Analyseur lexical I

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** (*token*) entier associé à la **catégorie lexicale** du mot accepté
- Un jeton (*token*) est généralement représenté par un entier positif ou une instance de classe
- 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), ...

Analyseur lexical II

- Pour les catégories lexicales variables, il faudra également “retourner” une **valeur sémantique** associée
- 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

Analyseur lexical III

- 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.

Analyseur lexical IV

- En reprenant l'exemple 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.
- 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 :

Analyseur lexical V

```
(INT,) (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 suivant décrit le fonctionnement d'un tel analyseur lexical

Algorithme Analyseur lexical I

Algorithme 2 : Analyseur lexical

Données : $B = (V, E, D = \{d\}, A, T)$; *JETON*[*A*] ; *flot* ;
Résultat : (Entier : le jeton reconnu, Chaîne : le lexème reconnu)
Fonction `analex`(*B, JETON[A], flot*) : (Entier, Chaîne)
début

```
    etat=d ; lexeme="" ; efinal=-1 ; lfinal=0 ; // Init.;
    tant que ((c=carSuivant(flot))≠ $) et (etat, c, e) ∈ T faire
        lexeme=lexeme . c ; etat=e;
        si e ∈ A alors
            efinal=e ; lfinal=|lexeme|;
    si etat ∈ A alors
        rejeter(flot, c) ; retourner (JETON[etat],lexeme);
    sinon
        si efinal > −1 alors
            rejeter(flot, c) ; rejeter(flot, sous-chaine(lexeme,lfinal,|lexeme|)) ; retourner
              (JETON[efinal], lexeme[0, lfinal − 1]);
        sinon
            si lexeme="" et c=$ alors
                retourner (0,"") ; // pas d'état final;
            sinon
                si lexeme="" alors
                    retourner (c,c);
                sinon
                    rejeter(flot, c) ; rejeter(flot, sous-chaine(lexeme,1,|lexeme|)) ;
                    retourner (lexeme[0], lexeme[0]) ;// tout sauf le 1er car;
```

Remarques sur l'algorithme I

- 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 !

Remarques sur l'algorithme II

- 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 !

Plan

Implémentation des analyseurs lexicaux I

- L'implémentation en C de la fonction d'analyse lexicale `analex()` correspondant à l'algorithme précédent suit.
- En C, une seule valeur pouvant être retournée par une fonction, on choisit de retourner le jeton et d'implémenter la valeur sémantique dans une variable globale `lexeme` de type chaîne de caractères
- On utilise l'AFD de la figure 1 et on transforme la définition de l'automate pour ajouter la définition des jetons dans un tableau entier JETON remplaçant le tableau FINAL (`afdJeton.h`) :

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

Nous représentons la fonction d'analyse lexicale `int analex()` dans le fichier `analexJeton.h` :

Implémentation des analyseurs lexicaux II

```
/**
 * @file analexJeton.h
 * @author Michel Meynard
 */
char lexeme[1024];  /* lexème courant de taille maxi 1024 */

int analex(){   /* reconnaît un mot sur l'entrée standard */
  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 */
```

Implémentation des analyseurs lexicaux III

```
  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];  /* ret le jeton correspondant */
}
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]; /* retourner le jeton */
}
else if (strlen(lexeme)==0 && c==EOF)
```

Implémentation des analyseurs lexicaux IV

```
    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 :

Implémentation des analyseurs lexicaux V

```
/** @file analexJeton.c
 * @author Michel Meynard
 */
#include <stdio.h>
#include <string.h>
#include "afdJeton.h"  /* Définition de l'AFD et des JETONS */
#include "analexJeton.h" /* Déf. fon : int analex() */

int main(){   /* Construction de l'AFD */
  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);
```

Implémentation des analyseurs lexicaux VI

```
  }
  return 0;
}
```

Après compilation de ce programme C, on l'exécute :

```
Saisissez un(des) mot(s) matchant a(b+c)?|bd suivi de EOF
↪  (CTRL-D) SVP : abdaabbc
Résultat : Jeton = 300 ; Lexeme = a
Résultat : Jeton = 302 ; Lexeme = bd
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 : baabc
Résultat : Jeton = 98 ; Lexeme = b
Résultat : Jeton = 300 ; Lexeme = a
Résultat : Jeton = 301 ; Lexeme = abc
Saisissez un(des) mot(s) matchant a(b+c)?|bd suivi de EOF
↪  (CTRL-D) SVP : xx
```

Implémentation des analyseurs lexicaux VII

```
Résultat : Jeton = 120 ; Lexeme = x
Résultat : Jeton = 120 ; Lexeme = x
Résultat : Jeton = 10 ; Lexeme =
```

Remarque : 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.

- Une dernière fonctionalité à réaliser par les analyseurs lexicaux est le **filtrage** des séparateurs (blancs : espaces, tabulations, ...) et des commentaires.
- dans notre implémentation précédente de l'exemple `analexJeton.h`, on fixera un jeton négatif pour les états finaux à filtrer

Implémentation des analyseurs lexicaux VIII

- Il suffira alors de modifier les retours de jeton négatif en appel récursif à analex() : `return JETON[etat];` deviendra alors `return (JETON[etat]<0 ? analex() : JETON[etat]);`.
- Idem pour `return JETON[efinal];`. On trouvera ces changements dans le fichier `analex.h` fourni pour les TD.
  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.

Plan

Un langage et un outil pour l'analyse lexicale : flex I

- 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 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

Un exemple I

Analyseur lexical de la figure 1 réécrit en flex :

```
%{                                          /* analflex.l */
 /* ZONE DE DEFINITION  (OPTIONNELLE) */
 /* 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 -1; /* ret un jeton pour tout le reste */}
%%
/* ZONE DES FONCTIONS C */
main()
{int j; char *invite="Saisissez un(des) mot(s) matchant
↪  a(b+c)?|bd suivi de EOF
 (CTRL-D) SVP : ";
 printf(invite);
```

Un exemple II

```
while ((j=yylex())!=0) printf("\nJeton : %i; de lexeme
↪  %s\n%s",j,yytext,invite);
}
```

Après compilation flex, `flex analflex.l`, puis compilation C et éditions de liens avec la bibliothèque flex, `gcc -o analflex lex.yy.c -lfl`, 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

Un exemple III

- 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.
- 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, d'autre part par l'ordre des expressions régulières dans le source lex

Un exemple IV

- 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.

Plan

Architecture d'un fichier flex I

Un source lex comprend 3 sections séquentielles :

- une section **optionelle** de définitions. Elle contient 2 sous-sections :
  - directives d'inclusions et définitions globales C (variables, types, ...) délimitées par %{ au début et %} à la fin
  - définitions flex (alias, conditions de démarrage, options, ...)
- Une section **obligatoire** de règles lex délimitée par %% au début. C'est la section centrale du source lex qui définit l'analyseur lexical en associant des instructions C à des expressions régulières.
- Une section 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 I

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 "}".

Syntaxe des expressions régulières I

Soit e et r deux ers quelconques, c et d deux caractères, m et n deux nombres entiers positifs :

Exemple	Signification	Opérateur
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	[cd]
[a-z] [0-9]	1 minuscule suivie d'1 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

Syntaxe des expressions régulières II

.	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$	interdit (1 seul opérateur contextuel)	
Bonjour/(toto)	Bonjour seulement si suivi par toto	e/r
<PHP>Dupont	Dupont seulement si on est dans la condition de démarrage PHP	<state>e
<<EOF>>	fin de fichier (seulement flex)	<<EOF>>
<state><<EOF>>	fin de fichier dans un certain condition de démarrage (seulement flex)	<s><<EOF>>
{chiffre}	chiffre = alias dans la 1ère section du source lex	{def}

Instruction(s) C I

- La partie droite de chaque règle est un bloc d'instructions C
- Le texte inclu entre accolades sera recopié intégralement dans `lex.yy.c` sans aucune analyse ni modification
- Les instructions C peuvent faire appel à des fonctions prédéfinies par lex ou définies par l'utilisateur dans la troisième section 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()`

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

- une autre solution pour `yywrap` consiste à utiliser dans le préambule l'option suivante :

Instruction(s) C II

```
%option noyywrap
```

- 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 section définitions : dans ce cas, la variable est globale ;
  - soit définie juste après l'accolade : dans ce cas, la variable est locale à la règle.

Un exemple I

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 I

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 I

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 section "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 ;

Fonctions prédéfinies II

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 condition de démarrage doit avoir été défini dans la première section grâce à `%Start etat` ou à `%x etat`. `BEGIN(0)` permet de revenir à l'condition de démarrage 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 I

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.

Ambiguités de correspondance II

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.

Ambiguités de correspondance III

- 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`.

Définitions flex : abbréviations ou alias I

Certains facteurs 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.
Utilisation :

```
{chiffre}+\.{chiffre}*(e[+-]?{chiffre}+)?   {return LITFLOT; /
```

Définitions flex : conditions de démarrage I

Une "Start condition" permet de conditionner la reconnaissance de certaines règles selon l'état dans lequel l'analyseur se trouve. Il permet ainsi de définir des sous-automates pour la reconnaissance de certaines parties de langages. Un exemple est la reconnaissance des balises PHP dans un document HTML. Une seule condition (INITIAL) est définie par défaut. La définition des conditions de démarrage se fait dans la première section :

```
%Start DANSCOM sectionRegle
%x PHP
```

Trois conditions de démarrage (start condition) sont définies ici. Avec l'instruction flex `%Start` (ou `%s`), les règles de la condition seront prioritaires mais les autres règles (sans condition) seront utilisées s'il n'y a pas de correspondance possible ! Il est donc préférable d'utiliser `%x PHP` (exclusif) si l'on veut que seules les règles de la condition soient utilisées. Ces conditions pourront être utilisées en préfixe des expressions régulières :

Définitions flex : conditions de démarrage II

```
"<?php "          {BEGIN(PHP);}
"/*"              {BEGIN(DANSCOM);}
<DANSCOM>"*/"     {BEGIN(INITIAL);}
<PHP>"?>"         {BEGIN(INITIAL);}
```

Définitions flex : options I

L'instruction `%option` permet de définir des options de compilation de l'analyseur lexical (qu'on aura pas besoin de spécifier dans la ligne de commlande). Par exemple :

```
%option noyywrap
```

Cette option indique que la fonction `yywrap()` n'est pas utilisée.
Autres options :

- c++ L'option `c++` permet de générer un analyseur lexical en C++ au lieu du C par défaut
- yylineno L'option `yylineno` permet de définir une variable entière globale `yylineno` qui sera incrémentée à chaque nouvelle ligne reconnue (utile pour les messages d'erreur)
- case-insensitive L'option `case-insensitive` permet de rendre insensible à la casse la reconnaissance des modèles (utile pour les langages HTML, SQL, ...)

Définitions C I

- Toute ligne de la section 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.
- Idem pour tout ce qui est inclus entre %{ et %}, ces délimiteurs étant détruits dans `lex.yy.c`. A part les variables globales, cette section permet d'inclure des macros `#include` `#define`, des `typedef`, ...

Troisième section I

- Cette section 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.
- Enfin, on peut utiliser des fichiers objets externes lors de l'édition de liens à condition d'avoir inclus leurs en-têtes dans la première section

Plan

La commande flex I

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 I

Voici la partie du `makefile` correspondant à la génération d'applications à partir de source `flex` d'extension `.l` sans la bibliothèque flex (il faut définir les fonctions `int main()` et `int yywrap()`.

```
.SUFFIXES:.l
CC=gcc
CFLAGS=-g
LEX=flex
.l:     # sans librairie
        @echo début $(LEX)-compil : $<
        $(LEX) $<
        @echo début compil C de lex.yy.c
        $(CC) $(CFLAGS) -o $* lex.yy.c
        @echo fin $(LEX)-compil : $<
        @echo Vous pouvez exécuter : $*
```

Plan

Actions C++ I

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;}
```

Actions C++ II

```
%%
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++ I

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

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

Plan

Liaison avec un analyseur syntaxique I

- 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()` et non plus `yylex()`
- Une ou plusieurs variables globales, `yylval` par exemple, peuvent être alors partagées par les 2 fonctions `yylex()` et `yyparse()`.
- Souvent la variable `yylval` sera utilisée pour stocker un attribut sémantique associé au jeton retourné par Flex
- La définition du type de `yylval` sera réalisé dans le fichier source bison qui sera le fichier maître du projet

Plan

Traduction des expressions régulières I

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

- 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
- Le principe revient à associer récursivement un automate à chaque noeud de l'arbre syntaxique de l'expression régulière

Traduction des expressions régulières II

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
    ;
```

Traduction des expressions régulières III

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**
    $B_g = (V, E_g, \{d_g\}, \{a_g\}, T_g)$ =`arbreVersAF(`$sag(a)$`)`;
    $B_d = (V, E_d, \{d_d\}, \{a_d\}, T_d)$ =`arbreVersAF(`$sad(a)$`)` ;
    retourner $B = (V, E_g \cup E_d, \{d_g\}, \{a_d\}, T_g \cup T_d \cup \{a_g \varepsilon d_d\})$;
    // *état fin. de* B_g *"fusionné" à l'état init. de* B_d

Traduction des expressions régulières IV

Algorithme 5 : construction d'un automate à partir d'un arbre (suite)

si *a est étiquetée par* $|$ **alors**

$B_g = (V, E_g, \{d_g\}, \{a_g\}, T_g)$ =`arbreVersAF(`*sag(a)*`)`
$B_d = (V, E_d, \{d_d\}, \{a_d\}, T_d)$ =`arbreVersAF(`*sad(a)*`)`
$B = (V, E_g \cup E_d \cup \{i, i+1\}, \{i\}, \{i+1\},$
$T_g \cup T_d \cup \{i\varepsilon d_g, i\varepsilon d_d, a_g\varepsilon i+1, a_d\varepsilon i+1\})$;
// on parallèlise B_g *et* B_d;
i=i+2;
retourner B ;

si *a est étiquetée par* $*$ **alors**

$B_g = (V, E_g, \{d_g\}, \{a_g\}, T_g)$ =`arbreVersAF(`*sous-arbre(a)*`)`;
$B = (V, E_g \cup \{i, i+1\}, \{i\}, \{i+1\},$
$T_g \cup \{i\varepsilon d_g, i\varepsilon i+1, a_g\varepsilon i+1, a_g\varepsilon d_g\})$
// on crèe un circuit sur B_g;
i=i+2;
retourner B;

Traduction des expressions régulières V

Quelques propriétés de l'algorithme de Thompson :

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

Traduction des expressions régulières VI

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.

Plan

Déterminisation I

On va écrire l'algorithme 6 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 6 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.

Algorithme de déterminisation I

Algorithme 6 : 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 ;

Remarques I

- 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 $\{\ldots e_{n+1} \ldots\}$ est final (Voir dans l'algorithme : $DA = \{Y \in DE / Y \cap A \neq \emptyset\}$).
- Remarquons que cette déterminisation permet de supprimer tous les chemins inaccessibles

Exemple I

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 : $\{1, 2\} \rightarrow 1; \{3, 4\} \rightarrow 2; \{4\} \rightarrow 3; D =$
$\{\{a, b\}, \{1..3\}, \{1\}, \{2, 3\}, \{1a2, 1b3\}\}.$

Plan

Minimisation I

- Rappelons que la forme canonique d'un langage régulier est son AFD minimal.
- l'algorithme de minimisation d'un AFD $B = (V, E, \{d\}, A, T)$ 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 de minimisation I

Algorithme 7 : 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;

Exemple I

Soit l' AFD complet suivant :

$$
\begin{aligned}
B = (\\
&V = \{a, b\},\\
&E = [1, 6],\\
&D = \{1\},\\
&A = \{3, 4, 5\},\\
&T = \{1a2, 1b3, 2a2, 2b3, 3a4, 3b6, 4a5, 4b6, 5a5, 5b6, 6a6, 6b6\}\\
)
\end{aligned}
$$

- 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 $\{1, 2, 6\}$)

Exemple II

- 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 :

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

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

Exercice

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

Plan

Introduction à l'analyse syntaxique I

- 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 : $X \rightarrow \alpha$ 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

Introduction à l'analyse syntaxique II

- Cette analyse peut se faire selon deux approches :
  - l'analyse syntaxique descendante consiste à partir de l'axiome qui constitue la racine de l'arbre de dérivation (ou arbre syntaxique)
  - l'arbre de dérivation est ainsi 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 alors 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

Introduction à l'analyse syntaxique III

- un arbre syntaxique représentant le programme. Celui-ci est soit un arbre de dérivation (arbre complet), soit un arbre abstrait (Abstract Syntax Tree) qui est un arbre simplifié. Cet arbre servira ensuite pour l'analyse sémantique puis la synthèse de la cible ou l'évaluation
- le programme cible directement compilé par la phase d'analyse syntaxique. On parle de traduction dirigée 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)

Plan

Analyse descendante récursive I

- méthode de programmation qui associe une fonction, souvent 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
- la grammaire doit posséder un certain nombre de propriétés pour permettre l'analyse descendante prédictive
- propriété fondamentale de ces grammaires : **non récursivité à gauche**
- 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

Analyse descendante récursive II

- le nombre de symboles terminaux nécessaires à la prédiction de la règle de production à choisir est une caractéristique des analyses descendantes prédictives
- si ce nombre est 0, on choisit une production quelconque et on tente la descente. Si celle-ci échoue : 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

Un exemple I

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

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

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.

Un exemple II

$G_{ETF} = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E, T, F\}, R, E)$ 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 étre tous générés avant de générer un opérateur *
- G_{ETF} étant récursive à gauche, on écrit une grammaire équivalente non récursive à gauche mais récursive à droite
- bien entendu, du point de vue syntaxique ces 3 grammaires sont équivalentes : $L(G_E) = L(G_{ETF}) = L(G_{ENR})$

Un exemple III

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

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

- il reste à écrire un vérificateur (reconnaisseur) syntaxique récursif utilisant un jeton de prédiction

Un exemple IV

- le programme C suivant (`analdesc.c`) effectue cette vérification syntaxique en calquant la structure de ses fonctions sur la grammaire G_{ENR}
- l'analyse lexicale est triviale car chaque lexème du langage est contitué d'un seul caractère (`getchar()`)
- on utilisera deux variables globales `jeton` et `numcar` pour conserver le jeton courant et la position dans la ligne
- il utilise des macros C (en majuscules) permettant de :
  - lire le jeton suivant (AVANCER()),
  - comparer le jeton courant avec le jeton attendu puis avancer (`TEST_AVANCE()`),
  - gérer les erreurs de syntaxe (`ERREUR_SYNTAXE()`)

Un exemple V

```
/**
 * analdesc.c vérifie la syntaxe d'une expression arith.
 * composée de nombres d'un chiffre, des opérations +, * et du
 * parenthésage.
 * L'expression est lue depuis l'entrée standard et se termine
 * par deux caractères EOF (Ctrl-D)
 * @author Michel Meynard
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
  // chaque macro est un bloc
#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);}
```

Un exemple VI

```
  // déclars en avant
void E(void);void R(void);void T(void);void S(void);void F(void);
↪

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

void E(void){
  T();              // règle : E->TR
  R();
}
void R(void){
  if (jeton=='+') {// règle : R->+TR
    AVANCER
    T();
    R();
  }
```

Un exemple VII

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

Un exemple VIII

```
    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;
```

Un exemple IX

```
}
```

Après compilation et édition de liens, on exécute ce vérificateur :

```
$ gcc -g -Wall -o analdesc analdesc.c
$ analdesc
1+2*3+(4+(5*(2+(1)+2)*3))^D
Mot reconnu
$ analdesc
4*81+2^D
Mot non reconnu : erreur de syntaxe au caractère numéro 4
```

Exercice I

Exercice

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

Une solution :

- Grammaire non ambiguë et non récursive à gauche : $S \rightarrow aSbS|\varepsilon$
- Programme C suivant :

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

#define AVANCER {jeton=getchar();numcar++;}
```

Exercice II

```
#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;
int numcar=0;

void S(void){       // AXIOME
  if (jeton=='a') { // règle : S->aSbS
    AVANCER
    S();
    TEST_AVANCE('b')
    S();
  }
  else ;            // règle : S->epsilon
}
int main(void){
```

Exercice III

```
  AVANCER             // initialiser jeton
  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;
}
```

Plan

Introduction I

- l'intérêt de la programmation récursive descendante réside dans son universalité (OS et langage)
- cependant une étude théorique des automates à pile est indispensable car le débogage des appels récursifs emboîtés devient vite compliqué
- 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 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

Introduction II

- 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 vues à la section précédente

Fonctionnement de l'automate à pile en analyse descendante I

Soit la grammaire $G = (V_T, V_N, R, S)$, chaque case de la table $M[V_N, V_T \cup \{\$\}]$ contient :

- soit une règle de production
- soit l'action ERREUR

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 (concordance),
- empile **à l'envers** la partie droite de la règle,
- termine en indiquant la réussite de l'analyse.

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

Fonctionnement de l'automate I

Algorithme 8 : Fonctionnement de l'automate

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

```
Pile=construirePileVide() ; empiler(Pile,$) ; empiler(Pile,S) ; jeton=lireFlot() ;
tant que vrai faire
    si sommet(Pile)=jeton et jeton=$ 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
                    dépiler(Pile) // remplaçons le non terminal
                    empiler(Pile, inverse(partieDroite(M[sommet(Pile), jeton]))) // de
                      droite à gauche
```

Un exemple simple I

Une grammaire de Dyck à un couple de parenthèses :
$G_D = (\{a, b\}, \{S\}, R, S)$ avec les règles de R suivantes :

$$S \rightarrow aSbS|\varepsilon$$

On obtient la table d'analyse suivante (voir algorithme 16) :

	a	b	\$
S	$S \rightarrow aSbS$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Etudions le fonctionnement de l'automate, c'est-à-dire :

- sa pile dont le fond est toujours le symbole \$ et qui grandit vers la gauche,
- son flot d'entrée, ici le mot `abaababb$`
- l'action réalisée à chaque étape en fonction du sommet de pile et du jeton courant situé à gauche du flot d'entrée

Un exemple simple II

Pile	Flot d'entrée	Action
S$	abaababb$	$S \rightarrow aSbS$
aSbS$	abaababb$	dépiler et avancer (a)
SbS$	baababb$	$S \rightarrow \varepsilon$
bS$	baababb$	dépiler et avancer (b)
S$	aababb$	$S \rightarrow aSbS$
aSbS$	aababb$	dépiler et avancer (a)
SbS$	ababb$	$S \rightarrow aSbS$
aSbSbS$	ababb$	dépiler et avancer (a)
SbSbS$	babb$	$S \rightarrow \varepsilon$
bSbS$	babb$	dépiler et avancer (b)
SbS$	abb$	$S \rightarrow aSbS$
aSbSbS$	abb$	dépiler et avancer (a)

Un exemple simple III

SbSbS$	bb$	$S \rightarrow \varepsilon$
bSbS$	bb$	dépiler et avancer (b)
SbS$	b$	$S \rightarrow \varepsilon$
bS$	b$	dépiler et avancer (b)
S$	$	$S \rightarrow \varepsilon$
$	$	Accepter

Un exemple simple IV

Plan

Algorithmique I

- 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.
- Les différents algorithmes suivants doivent parfois être utilisés pour cette mise en forme.

Suppression des ε-productions I

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 :

Suppression des ε-productions II

- $\exists n \in \mathbb{N}, E_n = E_{n+k}, \forall k \in \mathbb{N}$
- Par conséquent, $\forall X \in V_N, X \stackrel{*}{\Rightarrow} \varepsilon$ 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 :

Suppression des ε-productions III

- $G_{SE} = (V_T, V_N \cup \{S_1\}, R_1 \cup \{S_1 \to \varepsilon | S\}, S_1)$
- Sinon, $S \notin E_n$, on a $G_{SE} = G_{0E}$.

Un exemple I

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

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

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 :

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

Un exemple II

Pour finir, voici la grammaire équivalente à G et ne contenant qu'une ε-production :
$G_{SE} = (V_T, V_N \cup \{S_1\}, R_1 \cup \{S_1 \to \varepsilon | S\}, S_1)$.
Remarques

- 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 $(X \stackrel{+}{\Rightarrow} X)$.
- 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 I

- On suppose une grammaire sans ε-production.
- L'algorithme 9 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 : $X \rightarrow Y$.
- Seules les substitutions engendrant des cycles doivent être supprimées.
- Dans l'algorithme 9, 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.

Algorithme de Suppression des cycles I

Algorithme 9 : 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 $X_i \in V_N$:
$FTS(X_i) = \{X_j \in V_N / X_i \overset{+}{\Rightarrow} X_j\}$
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**
      $R_{SC} = R_{SC} \cup \{X_i \to \alpha\}$ /* *recopier* */
  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* */

Remarques I

- preuve de l'élimination des cycles : les seules règles de substitutions $(X_i \rightarrow X_j)$ autorisées dans R_{SC} impliquent que X_i et X_j ne soient pas dans le même cycle
- 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 I

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

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

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} :

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

Exemple II

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

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

Suppression de la récursivité à gauche immédiate I

- Une récursivité à gauche immédiate d'un symbole non terminal X se matérialise par au moins une règle de production $X \rightarrow X\alpha$
- 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 10 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 I

Algorithme 10 : 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**
  $P' = P' \cup \{R_X \to \alpha_j R_X\}$

Remarques I

- L'algorithme 10 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 I

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

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

Après application de l'algorithme 10, on obtient la grammaire suivante :
$G_{ENRI} = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E, R_E\}, P', E)$ avec les règles de P' suivantes :

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

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

Exercice I

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

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

Supprimer la récursivité à gauche dans cette grammaire !

Exercice II

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

Suppression de la récursivité à gauche I

- 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 $X_1 \to X_2 a | a, X_2 \to X_1 b | b$ il n'y pas de récursivité à gauche immédiate mais il y a de la récursivité à gauche !
- L'algorithme 11 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 $X \stackrel{+}{\Rightarrow} X\alpha$.

Algo. de suppression de la récursivité à gauche I

Algorithme 11 : 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**

- $P = \{X_i \to \gamma \in R\}$ // *ensemble des productions* $X_i \to \ldots$
- **tant que** $\exists X_i \to X_j\alpha \in P$ *telle que* $i > j$ **faire**
  - $P = P - \{X_i \to X_j\alpha\}$ // *suppression*
  - **pour chaque** *production* $X_j \to \beta \in R_{NR}$ **faire**
    - $P = P \cup \{X_i \to \beta\alpha\}$ // *remplacement*
- P'=Supprimer la récursivité immédiate dans P (algo. 10)
- $R_{NR} = R_{NR} \cup P'$

Remarques I

- 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 $i \geq j$
- 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
- après la dérécursivation, on obtient souvent des grammaires ayant des ε-productions

Remarques II

- Ainsi, dans l'exemple de la page 114, la grammaire G_{ENR} est non récursive à gauche et contient des ε-productions et ceci n'est pas génant
- En effet, ces productions ne peuvent en aucun cas impliquer une récursivité à gauche

Exemple I

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

$$\begin{aligned} X_1 &\rightarrow X_2 a | d \\ X_2 &\rightarrow X_3 a | X_1 b \\ X_3 &\rightarrow X_1 a \end{aligned}$$

Après application de l'algorithme 11, 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 :

Exemple II

$$
\begin{aligned}
X_1 &\rightarrow X_2 a | d \\
X_2 &\rightarrow X_3 a R_2 | d b R_2 \\
R_2 &\rightarrow \varepsilon | a b R_2 \\
X_3 &\rightarrow d b R_2 a a R_3 | d a R_3 \\
R_3 &\rightarrow \varepsilon | a R_2 a a R_3
\end{aligned}
$$

Factorisation à gauche I

- 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 faudrait donc pouvoir lire plusieurs jetons en avance !
- La factorisation des parties droites est destinée à réduire à 1 ce nombre de jetons de prévision
- les grammaires ainsi formées seront qualifiée de $LL(1)$ (Left to right scanning of the input, Leftmost derivation, 1 look-ahead symbol)

Algorithme de factorisation I

Algorithme 12 : 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 du plus grand nombre de 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 du plus grand nombre de parties droites des X-productions de R_F
  marquer X

Exemple I

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 12, 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.

La fonction premiers() I

- 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 13 réalise cette fonction.

Algorithme premiers(X) I

Algorithme 13 : 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* $X \to Y_1 Y_2 \ldots Y_k \alpha$ *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

Remarques I

- L'algorithme 13 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
- cet algorithme ne peut être utilisé sur une grammaire récursive à gauche (appel récursif infini)
- Cette propriété reste fondamentale pour le calcul des `premiers`(α) qui fait appel aux `premiers`(X)
- L'algorithme 14 calcule justement ces `premiers`(α).

Exemple I

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 :

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

Exemple II

On obtient par l'application de l'algorithme 13 :

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

Algorithme Premiers(mot) I

Algorithme 14 : 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*
  $Resultat = Resultat \cup (\texttt{premiers}(Y_1) - \{\varepsilon\})$
  i=1
  tant que $i \leq k$ *et* $\varepsilon \in \texttt{premiers}(Y_i)$ **faire**
    $i = i + 1$
    $Resultat = Resultat \cup (\texttt{premiers}(Y_i) - \{\varepsilon\})$ // *non réc. gauche*
  si $i = k + 1$ **alors**
    $Resultat = Resultat \cup \{\varepsilon\}$ // *tous les Y_i s'effacent*
  retourner Resultat

Suivants I

- L'algorithme 15 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 :
  $TabSuivants[X_i] = \{x \in V_T \cup \{\$\} / S \stackrel{*}{\Rightarrow} \alpha X_i x \beta\}$
- L'algorithme 15 calcule ce tableau $TabSuivants[X_i]$

Algorithme Suivants I

Algorithme 15 : Suivants

Données : $G = (V_T, V_N = \{X_1, X_2, \ldots, X_n\}, R, X_1)$, une grammaire
Résultat : un tableau $TabSuivants[X_i]$ d'ensembles de terminaux $\{x_1, x_2, \ldots, x_m\} \subseteq (V_T \cup \{\$\})$

```
TabSuivants[X1] = {$} // initialisation pour l'axiome
pour i=2 à n faire
    TabSuivants[Xi] = ∅ // initialisation

répéter
    stable=vrai // booléen testant la stabilité du tableau
    pour chaque production Y → γ de R faire
        pour chaque non terminal X de γ : Y → αXβ avec γ = αXβ faire
            si β = ε alors
                si TabSuivants[Y] ⊈ TabSuivants[X] alors
                    stable=faux
                    TabSuivants[X] = TabSuivants[X] ∪ TabSuivants[Y]
            sinon
                si premiers(β) − {ε} ⊈ TabSuivants[X] alors
                    stable=faux
                    TabSuivants[X] = TabSuivants[X] ∪ (premiers(β) − {ε})
                si ε ∈ premiers(β) // β est effaçable alors
                    si TabSuivants[Y] ⊈ TabSuivants[X] alors
                        stable=faux
                        TabSuivants[X] = TabSuivants[X] ∪ TabSuivants[Y]
jusqu'à stable;
```

Exemple I

Soit la grammaire non récursive à gauche G_{ENR} de l'exemple de la page 165. On obtient par l'application de l'algorithme 15 :

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

Construction de la table d'analyse I

- L'algorithme 16 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 a un choix à réaliser !
- Ceci n'est pas envisageable pour des raisons d'efficacité (backtrack)
- il sera alors nécessaire de transformer la grammaire

Algorithme de construction de la table d'analyse I

Algorithme 16 : 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 $x \in \texttt{premiers}(\alpha) - \{\varepsilon\}$ **faire**
    $M[X,x] = M[X,x] \cup X \to \alpha$
  si $\varepsilon \in \texttt{premiers}(\alpha)$ **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 I

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

	a	b	\$
S	$S \rightarrow aSbS$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Exemple G_{ENR} I

Reprenons une grammaire non récursive à gauche plus complexe et voyons la table d'analyse générée
$G_{ENR} = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E, R, T, S, F\}, X, E)$ avec les règles de X suivantes :

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

Exemple G_{ENR} II

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

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

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

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

On obtient finalement par l'application de l'algorithme 16, la table suivante :

Exemple G_{ENR} III

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

Les cases vides correspondent à des erreurs de syntaxe !

Contre-exemple I

On choisit de placer un ensemble de productions et pas seulement une production, dans l'algorithme 16 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
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 :

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

Contre-exemple II

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

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

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

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

On obtient finalement par l'application de l'algorithme 16, la table suivante :

Contre-exemple III

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

- Dans cette table, l'entrée $M[S', e]$ contient deux productions possibles
- Il faut, dans ce cas, choisir de conserver la production $S' \to eS$ 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 résolue 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.

Contre-exemple IV

- Attention, cet exemple est particulier et ne peut fonctionner dans le cas général où un choix produira un langage reconnu plus petit !

Plan

Grammaires LL(1) I

Définition

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.

Grammaires LL(1) II

Théorème

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

Théorème

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).

Plan

Conclusion sur l'analyse descendante I

- 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) (exemple de la grammaire `if then else` qui génère 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 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 !

Conclusion sur l'analyse descendante II

- D'un point de vue plus pratique, le problème principal des grammaires LL(1) résulte du 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 syntaxiques deviennent difficiles à mettre en oeuvre
- L'analyse syntaxique ascendante va nous permettre de conserver des grammaires récursives à gauche et/ou à droite plus intuitives

Plan

bison, un outil pour l'analyse syntaxique ascendante I

- 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 sur tous les OS

bison, un outil pour l'analyse syntaxique ascendante II

- Bien entendu, bison peut être utilisé conjointement à flex qui fournira lui les jetons consommés par l'analyseur syntaxique généré par bison

Un exemple I

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 bison, reconnaissant les mots du langage généré par cette grammaire.
Voici le source bison obtenu :

```
%{                                          /* veriflog.y  */
  #include  <stdio.h>
  int yylex(void);  void yyerror(char *s);
%}
%%
expr    :       '(' expr ')'
        {}
        |       expr '|' expr
```

Un exemple II

```
        {}
        |       expr '&' expr
        {}
        |       '!' expr
        {}
        |        '0'
        {}
        |       '1'
        {}
        ;
%% /* début des fonctions C */
int yylex(void) {  // analyseur lexical filtrant les blancs
  int c;
  while(((c=getchar())==' ') || (c=='\t'))
    ;
  return (c);
}
```

Un exemple III

```
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 bison
    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 :

Un exemple IV

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

Un exemple V

- 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`.

Plan

Architecture I

Un source bison comprend 3 parties séquentielles :

- une partie déclaration contenant des déclarations C contenues entre %{ et %}, et des déclarations spécifiques à bison.
- 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 bison 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 bison I

- Une règle bison 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 I

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 bison à l'aide du déclarateur bison : `%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 bison
3. Inclure ce fichier dans la partie définition du source flex.

Symboles terminaux (jetons) et non terminaux II

Bien entendu, si l'on n'utilise pas flex, 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'))
```

Symboles terminaux (jetons) et non terminaux III

```
    ;
  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

Symboles terminaux (jetons) et non terminaux IV

- 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 I

Exercice

Ecrire le source bison de vérification du langage de Dyck

```
%{#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 */
```

Exercice II

```
    printf("\nMot de Dyck reconnu\n");
  else
    printf("\nMot non reconnu\n");
  return 0;
}
```

Partie droite de règle I

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 bison, cependant il est fortement recommandé d'écrire des grammaires récursives à gauche pour optimiser le fonctionnement de l'analyseur

Valeur sémantique ou attribut I

Associée à chaque symbole, terminal ou non, une **valeur sémantique** (attributs des grammaires attribuées) est définie automatiquement par `bison`. 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 flex 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 bison `union`. Par exemple,

Valeur sémantique ou attribut II

```
%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

Valeur sémantique ou attribut III

- 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;`
- En C++, les champs de l'union devront être de type pointeur sur classe

Actions I

- N'importe quelle instruction C peut apparaître dans un bloc d'actions
- De plus, `bison` 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`

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

Actions II

```
%{                                          // interlog.y
#include   <stdio.h>
#define YYSTYPE int                       /* inutile */
int yylex(void);  void yyerror(char *s);
%}
%%
liste   :                       {}
        | liste ligne           {}
        ;
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;}
```

Actions III

```
        ;
%%
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 :

Actions IV

```
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 I

- 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 `$i`
- En effet, un bloc intermédiaire est lui-même associé à un attribut `$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 $$

Actions à l'intérieur de la partie droite II

- 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 '}'
    ;
```

Actions à l'intérieur de la partie droite III

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

Actions prédéfinies I

$$	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 ;

Actions prédéfinies II

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 première partie : déclarations I

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 bison 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 ....

La première partie : déclarations II

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 I

- Dans la partie déclaration, on peut définir des **priorités** 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 bison par `%nonassoc`
- La déclaration de l'associativité à gauche est effectuée par : `%left JETONOP1 JETONOP2 JETONOP3 ...` où `JETONOPi` est un jeton nommé ou non d'opérateur

Associativité et priorité des opérateurs II

- 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, est définie simplement par l'odre 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 de priorités I

```
%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`

Exemple de priorités II

- 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 I

- `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 `bison`
- La valeur sémantique (attribut) d'un jeton sera passée de `flex` à `bison` par l'intermédiaire de la variable `yylval` qui est de type `YYSTYPE`

Divers I

Débogage : afin de déboguer l'analyseur syntaxique, il suffit de positionner la variable `bison` prédéfinie `yydebug` à 1 avant l'appel à `yyparse()` ou pendant son exécution

Makefile :

```
YACC=bison
YACCFLAGS=-ydtv
#-y 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 : $*
```

Divers II

On peut également utiliser du C++ dans les actions. Soi, par exemple, le source suivant :

```
%{
#include <iostream>
using namespace std;
int yylex(void);void yyerror(char *s);
class A{
public:
  void essai(int n){cout<<"Suite de "<<n<<"
  ↪  identificateurs"<<endl;
  }
};
%}
%token IDENTIF
%%
liste   : {/* chaine vide sur fin de fichier Ctrl-D */}
        | liste ligne {}
```

Divers III

```
        ;
ligne   : '\n'                  {/* ligne vide : expression vide
↪  */}
        | error '\n'            {yyerrok; /* synchro après la fin de
        ↪  ligne */}
        | expr '\n'             {A a;a.essai($1);}
        ;
expr    : IDENTIF                   {$$=1;}
        | expr IDENTIF              {$$++;}
        ;
%%
int yylex() {
  int c;
  while(((c=getchar())==' ') || (c=='\t')) ; /* filtrage des
  ↪  blancs */
  if ((c<='z') && (c>='a')) {
    while(((c=getchar())<='z') && (c>='a')) ;
```

Divers IV

```
    ungetc(c,stdin);
    return (IDENTIF);
  }
  return c;          /* erreur lexicale */
}
void yyerror(char *s) {cerr<<s<<endl;}
int main(){yydebug=0; return yyparse();}
```

Ce source pourra être compilé par l'entrée de makefile suivante :

```
CPP=g++
CPPFLAGS=-g
.y+:
  @echo debut $(YACC)-compil : $<
  $(YACC) $(YACCFLAGS) $<
  @echo debut compil c++ avec edition de liens de y.tab.c
  $(CPP) $(CPPFLAGS) -o $* y.tab.c
```

Divers V

```
@echo fin $(YACC)-compil de : $<
@echo Vous pouvez executer : $*
```

Plan

Un exemple bison complet : une calculette I

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

```
%{                                          /* 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
%%
[ \t]+              {/* filtrer les blancs */}
{entier}|{entier}\.{chiffre}*|{chiffre}*\.{entier} {
                    /* laisser l'accolade à la ligne precedente */
```

Un exemple bison complet : une calculette II

```
                    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 !
↪   */}
%%
```

Un exemple bison complet : une calculette III

Voici le source bison :

```
%{                                          /* calc.y  */
#include <math.h>
int errSemantiq=0;                          /* vrai si erreur sémantique :
↪  */
#define DIVPARO 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 '^'
```

Un exemple bison complet : une calculette IV

```
%%

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 */
          }
```

Un exemple bison complet : une calculette V

```
    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");
```

Un exemple bison complet : une calculette VI

```
        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   {
```

Un exemple bison complet : une calculette VII

```
  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 ... */
```

Un exemple bison complet : une calculette VIII

```
        }
        |       PI                  {$$ = M_PI;}
        |       LITFLOT             {$$ = $1;}
        ;
%%
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(void){yydebug=0; return yyparse();}
```

Un exemple bison complet : une calculette IX

Voici l'entrée de makefile :

```
calc :          calc.y calc.l
  # d'abord yacc pour y.tab.h puis lex puis gcc
  @echo debut $(YACC)-compil : calc.y
  $(YACC) $(YACCFLAGS) calc.y
  @echo debut $(LEX)-compil : calc.l
  $(LEX) calc.l
  @echo debut compil c avec edition de liens
  $(CC) -g -Wall  -o calc y.tab.c lex.yy.c -lm    # lib math
  @echo fin compil : vous pouvez executer calc
```

Un exemple bison complet : une calculette X

Voici une exécution :

```
$ make calc
$ calc
2+3*4
Résultat :      14.00
-5--2^2
Résultat :      -1.00
1/4-1/2^2
Résultat :       0.00
1+3*2^3^(ln(100)/ln(10))
Résultat :    1537.00
```

Plan

Analyse ascendante par automate à pile I

- Nous allons étudier l'analyse ascendante et plus particulièrement l'analyse LALR utilisée dans bison.
- 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.

Analyse ascendante par automate à pile II

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

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

- Considérons le mot d'entrée 1+2+3$
- L'analyse du mot commence sur le 1 (Left to right scanning)
- Ce symbole 1 est décalé du flot de jeton sur la pile (opération Shift)
- Puis la règle $E \rightarrow 1$ est appliquée en réduisant sa partie droite (1) dans la pile et en la remplaçant par sa partie gauche E (opération Reduce)
- Arrivé sur le $+$, l'analyseur empile ce symbole car il ne peut pas appliquer de réduction

Analyse ascendante par automate à pile III

- Le 2 est ensuite reconnu et décalé sur la pile
- Puis, 2 est réduit en E ($E \rightarrow 2$) dans la pile
- On s'aperçoit qu'on peut alors réduire (Reduce) le mot sur la pile (E+E) en appliquant la règle $E \rightarrow E + E$
- La pile ne contient donc plus que E
- En continuant le même procédé, on reconnait les productions $E \rightarrow 3$ puis $E \rightarrow E + E$
- On a donc la dérivation droite, obtenue à l'envers :
  $E \overset{1}{\Rightarrow}_{E \rightarrow E+E} E + E \overset{1}{\Rightarrow}_{E \rightarrow 3} E + 3 \overset{1}{\Rightarrow}_{E \rightarrow E+E} E + E + 3 \overset{1}{\Rightarrow}_{E \rightarrow 2}$
  $E + 2 + 3 \overset{1}{\Rightarrow}_{E \rightarrow 1} 1 + 2 + 3$
- 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

Analyse ascendante par automate à pile IV

- Cet analyseur choisit l'action Reduce sur un conflit Shift/Reduce
- `bison`, 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 !

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

Définition

Un manche d'un mot (pas forcément terminal) $m = \alpha\beta\gamma$ est un couple constitué :

- d'une production $X \rightarrow \beta$,
- d'une position p dans m telle que $m[p, p + |\beta|[= \beta$;

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

- Dans l'exemple précédent, le mot 1+2+3 ne possède qu'un manche ($E \rightarrow 1,1$)
- En effet, ni ($E \rightarrow 3,5$), ni ($E \rightarrow 2,3$) ne sont des manches car ni 1+2+E, ni 1+E+3 ne dérive de E par une dérivation droite

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

- Par contre, E+E+3 possède deux manches : $(E \rightarrow E + E, 1)$ et $(E \rightarrow 3, 5)$
- On peut donc choisir entre les deux réductions possibles
- Dans l'exemple précédent, 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 cet exemple

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

- Il peut exister d'autres types de conflits Reduce/Reduce lorsque deux manches sont réductibles l'un étant suffixe de l'autre
- 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

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.

Table d'analyse d'un analyseur LR I

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 $X \rightarrow \alpha$. 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 : $R(X \rightarrow \alpha)$.
  - 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.

Table d'analyse d'un analyseur LR II

- 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 17.

Algorithme de Fonctionnement de l'automate I

Algorithme 17 : 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
```

Exemple I

Pour illustrer le fonctionnement de l'algorithme 17, prenons un exemple simple d'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 :

$$S \rightarrow SaSb|\varepsilon$$

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

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

Exemple II

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($S \to SaSb$)

Exemple III

\$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

Exemple IV

Ce qui donne la dérivation droite suivante :

$S \overset{1}{\Rightarrow} SaSb \overset{1}{\Rightarrow} SaSaSbb \overset{1}{\Rightarrow} SaSabb \overset{1}{\Rightarrow} SaSababb \overset{1}{\Rightarrow}$
$Saababb \overset{1}{\Rightarrow} SaSbaababb \overset{1}{\Rightarrow} Sabaababb \overset{1}{\Rightarrow} abaababb$

Plan

SLR et LALR I

- 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 utilisé par bison, résolvant certains problèmes de SLR au prix d'une taille plus importante des tables

SLR et LALR II

- 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 [2] pour ceux qui souhaiteraient en savoir plus ...

Plan

Construction de la collection canonique SLR I

Définition

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 $G_d = (\{a, b\}, \{S\}, R = \{S \rightarrow SaSb|\varepsilon\}, S)$
- L'ensemble des items de G est $Items(G) = \{S \rightarrow .SaSb, S \rightarrow S.aSb, S \rightarrow Sa.Sb, S \rightarrow SaS.b, S \rightarrow SaSb., S \rightarrow \varepsilon.\}$
- 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

Construction de la collection canonique SLR II

- 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

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 $S' \rightarrow S$: $G' = (V_T, V_N \cup \{S'\}, R \cup \{S' \rightarrow S\}, S')$

- 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.

Construction de la collection canonique SLR III

- 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 18

Algorithme Fermeture d'un ensemble d'items I

Algorithme 18 : 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)

Exemple I

Le principe de l'algorithme 18 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.
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 $\{S \to Sa.Sb\}$.
$Fermeture(\{S' \to .S\}) = \{S' \to .S, S \to .SaSb, S \to \varepsilon.\}$
$Fermeture(\{S \to Sa.Sb\}) = \{S \to Sa.Sb, S \to .SaSb, S \to \varepsilon.\}$

Collection canonique I

- 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 19 décrit cette construction.

Algorithme Construction de la collection canonique I

Algorithme 19 : 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*
$E = \{Fermeture(\{S' \rightarrow .S\})\}$ // *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* $Y \rightarrow \alpha.x\beta \in I$ **faire**
    $transition(I, x) = Fermeture(\{Y \rightarrow \alpha x.\beta\})$ // *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*;

Remarques et exemple I

- Remarquons que l'algorithme 19 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

Remarques et exemple II

Exemple :
Soit la grammaire de Dyck augmentée :
$G' = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$. Calculons l'automate correspondant :

$I_0 = Fermeture(\{S' \to .S\}) = \{S' \to .S, S \to .SaSb, S \to \varepsilon.\}$
$I_1 = Fermeture(\{S' \to .S, S \to .SaSb\}) = \{S' \to S., S \to S.aSb\}$
$T = \{(I_0, S, I_1)\}$
$I_2 = Fermeture(\{S \to Sa.Sb\}) = \{S \to Sa.Sb, S \to .SaSb, S \to \varepsilon.\}$
$T+ = \{(I_0, S, I_1), (I_1, a, I_2)\}$
$I_3 = Fermeture(\{S \to SaS.b, S \to S.aSb\}) = \{S \to SaS.b, S \to S.aSb\}$
$T+ = \{(I_0, S, I_1), (I_1, a, I_2), (I_2, S, I_3)\}$
$I_4 = Fermeture(\{S \to SaSb.\}) = \{S \to SaSb.\}$
$I_2 = Fermeture(\{S \to Sa.Sb\}) = \{S \to Sa.Sb, S \to .SaSb, S \to \varepsilon.\}$

Remarques et exemple III

$T+ = \{(I_0, S, I_1), (I_1, a, I_2), (I_2, S, I_3), (I_3, b, I_4), (I_3, a, I_2), \}$

- 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

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

Remarques et exemple IV

- 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

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.

Remarques et exemple V

Exemple

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 $S \to \varepsilon$.

Plan

Construction de la table Action SLR I

Algorithme 20 : 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 $i = S' \rightarrow S.$ **faire**
        ajouter "Accepter" à $Action[I_j, \$]$
      cas où $i = X \rightarrow \alpha.a\beta$ *avec* $a \in V_T$ *et* $(I_j, a, I_k) \in T$ **faire**
        ajouter Shift I_k à $Action[I_j, a]$
      cas où $i = X \rightarrow \alpha.$ *et* $i \neq S' \rightarrow S.$ **faire**
        pour chaque $x \in TabSuivants[X]$ **faire**
          ajouter $Reduce(X \rightarrow \alpha)$ à $Action[I_j, x]$
      cas où *autres* **faire**
        ne rien faire
  pour chaque *case vide* $Action[I_j, x]$ **faire**
    écrire "Erreur" dans $Action[I_j, x]$

Remarques I

- Remarquons qu'une seule action Accepter existe qui correspond à la réduction $S' \rightarrow S$ 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

Pour appliquer l'algorithme 20 sur la grammaire de Dyck augmentée $G' = (\{a, b\}, \{S, S'\}, \{S \rightarrow SaSb|\varepsilon, S' \rightarrow S\}, S')$, il nous faut calculer les suivants de S : $TabSuivants[S] = \{a, b, \$\}$. On obtient alors la table suivante :

Remarques II

	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($S \to SaSb$)

On peut maintenant écrire l'algorithme 21 de construction de la table Successeur SLR.

Construction de la table Successeur I

Algorithme 21 : 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 $Successeur[I_j, X]$

Remarques et exemple I

- 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.
- L'algorithme 21 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 I

Théorème

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

Théorème

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 : $G_g = (\{a, b\}, \{S, S'\}, \{S \rightarrow SaSb|\varepsilon, S' \rightarrow S\}, S')$
- Il existe une autre grammaire SLR engendrant le même langage : $G_d = (\{a, b\}, \{S, S'\}, \{S \rightarrow aSbS|\varepsilon, S' \rightarrow S\}, S')$.

Efficacité II

- La grammaire de Dyck récursive à gauche engendrera un automate dont la pile grossira moins que celle de l'automate "à droite" car les réductions pourront s'effectuer dès que possible
- C'est la raison pour laquelle on privilégie les grammaires récursives à gauche qui correspondent en plus au fonctionnement habituel des opérateurs majoritairement associatifs à gauche !

Un exercice I

Exercice

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

Un exercice II

Solution : AFD :

$I_0 = Fermeture(\{S' \rightarrow .S\}) = \{S' \rightarrow .S, S \rightarrow .aSbS, S \rightarrow \varepsilon.\}$

$T = \{(I_0, a, I_1)\}$

$I_1 = Fermeture(\{S \rightarrow a.SbS\}) = \{S \rightarrow a.SbS, S \rightarrow .aSbS, S \rightarrow \varepsilon.\}$

$T+ = \{(I_1, a, I_1)\}$

$T+ = \{(I_1, S, I_2)\}$

$I_2 = Fermeture(\{S \rightarrow aS.bS\}) = \{S \rightarrow aS.bS\}$

$T+ = \{(I_2, b, I_3)\}$

$I_3 = Fermeture(\{S \rightarrow aSb.S\}) = \{S \rightarrow aSb.S, S \rightarrow .aSbS, S \rightarrow \varepsilon.\}$

$T+ = \{(I_3, a, I_1)\}$

$T+ = \{(I_3, S, I_4)\}$

$I_4 = Fermeture(\{S \rightarrow aSbS.\}) = \{S \rightarrow aSbS.\}$

$T+ = \{(I_0, S, I_5)\}$

$I_5 = Fermeture(\{S' \rightarrow S.\}) = \{S \rightarrow S.\}$

Un exercice III

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($S \to aSbS$))

Un exercice IV

- 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 $S \rightarrow aSbS$ 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

Plan

Les conflits et leur résolution par bison I

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

Nous pouvons maintenant construire un morceau de la table Action :

Les conflits et leur résolution par bison II

	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 algorithmiques existent pour les grammaires LR(1) dont la méthode LALR de bison. L'option `-v` de bison permet notamment de visualiser les tables d'analyse utilisées. Voici, par exemple, le fichier

Les conflits et leur résolution par bison III

.output obtenu avec la grammaire
$G_g = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$.

```
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
```

Les conflits et leur résolution par bison IV

```
    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 l'exemple du transparent 251.

Conflit Shift/Reduce I

Que fait bison lorsqu'il rencontre des conflits ? Sur conflit Shift/Reduce, bison 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 bison fournit un analyseur privilégiant le décalage du "else" plutôt que la réduction du iEtS empilé. Voici la partie descriptive fournie par `bison -v` :

Conflit Shift/Reduce II

```
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 bison choisit d’utiliser la première règle dans l’ordre de description de la grammaire du source bison. Il est extrèmement périlleux d’utiliser cette caractéristique dans un analyseur car

Conflit Shift/Reduce III

l'ordre des règles de production dans le source bison peut souvent varier dans la phase de conception du langage.

Conflits multiples I

Un autre exemple de gestion des conflits dans bison 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)
```

Conflits multiples II

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

- L'action Shift a bien été privilégiée par rapport aux deux reduce possibles
- bison 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 certaines grammaires LR(1), c'est-à-dire nécessitant un seul jeton de prévision, ne sont pas analysables avec la méthode LALR.

Outils d'analyse des conflits I

- Bison avec l'option `verbose -v` fournit un fichier texte d'extension `output` vu auparavant
- avec l'option `graph -g`, il fournit également un graphe `graphviz` d'extension `gv` illustrant la collection canonique, les conflits et leur résolution !
- Le graphe du transparent suivant a été obtenu par la commande :

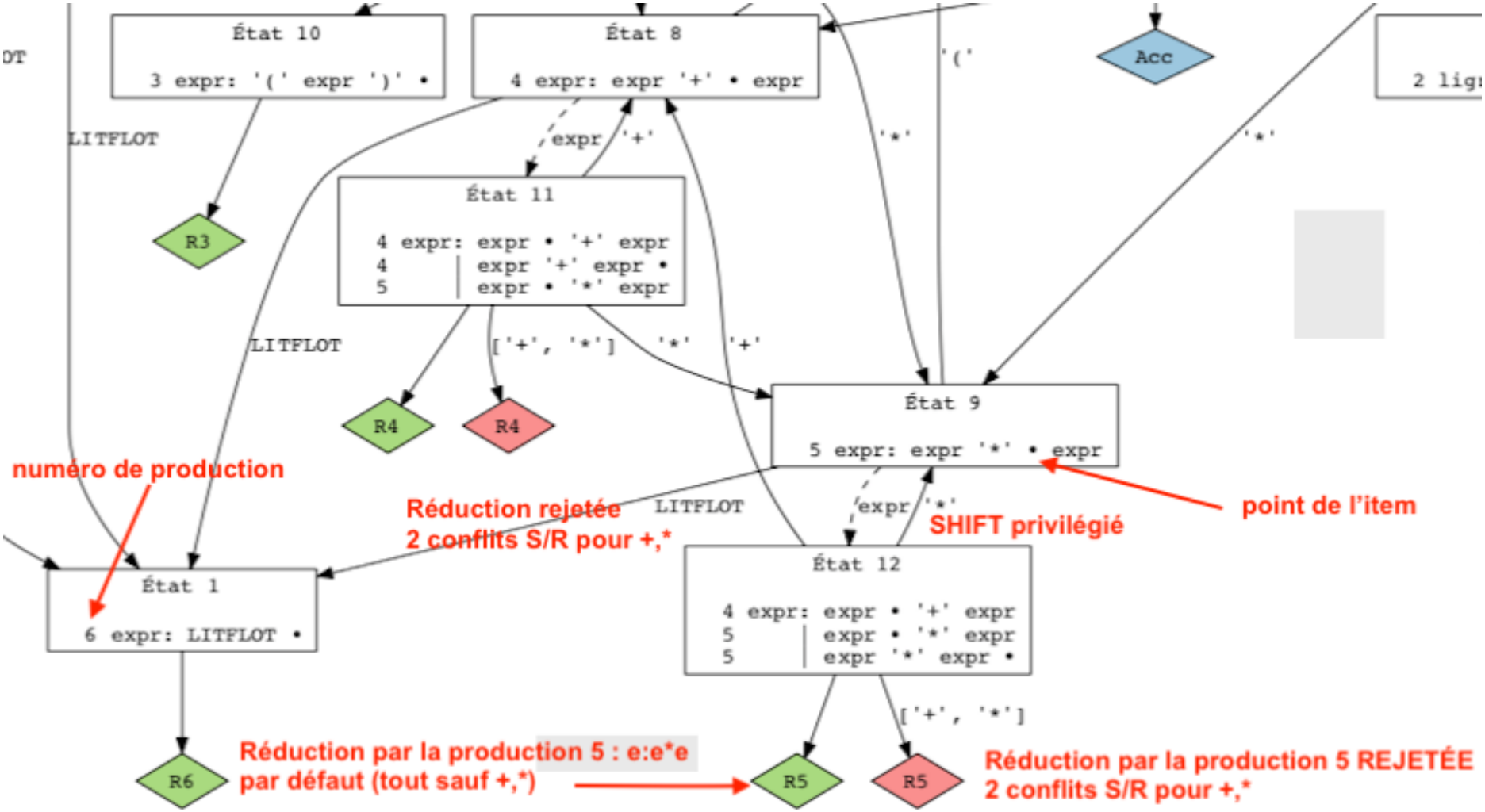
```
$ bison -yvg calcAvecConflits.y
```

- La grammaire `calcAvecConflits.y` contient :

```
expr    :       '(' expr ')'    {$$ = $2;}
|       expr '+' expr   {$$ = $1 + $3;}
|       expr '*' expr   {$$ = $1 * $3;}
|       LITFLOT         {$$ = $1;}
;
```

- Elle génère 4 conflits S/R !

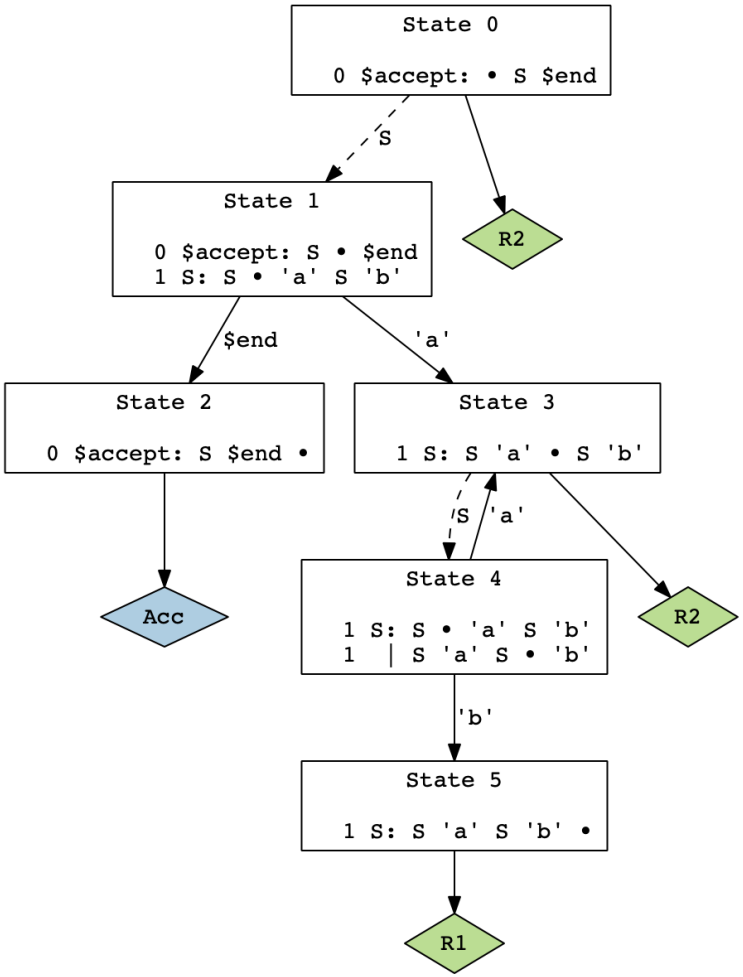
Outils d'analyse des conflits II



La grammaire de Dyck I

- Ci-après le graphe de la grammaire de Dyck
  $G' = (\{a, b\}, \{S, S'\}, \{S \to SaSb|\varepsilon, S' \to S\}, S')$ obtenu par bison
- Cette grammaire est SLR, il n'y a donc aucun conflit
- On peut remarquer que les préfixes viables sont représentés par les chemins de l'automate
- Les états (ensembles d'items) ne sont pas fermés transitivement

La grammaire de Dyck II



Plan

Introduction à l'analyse sémantique I

Nous allons ici étudier une certain nombre de techniques concernant l'analyse sémantique de code source :

- après l'analyse syntaxique qui a produit un AST (Abstract Syntax Tree), l'analyse sémantique à comme missions :
  - la résolution des noms (en liaison avec une table des symboles)
  - la vérification des types (types mismatch)
  - d'autres vérifications spécifiques au langages (nombre et types des paramètres d'une fonction)
- quelques généralités, beaucoup de spécificités liées au langage source
- les grammaires attribuées, une théorie pour la traduction ou l'interprétation dirigée par la syntaxe
- mise en pratique des grammaires attribuées avec `bison`

Plan

AST ou Arbre syntaxique abstrait I

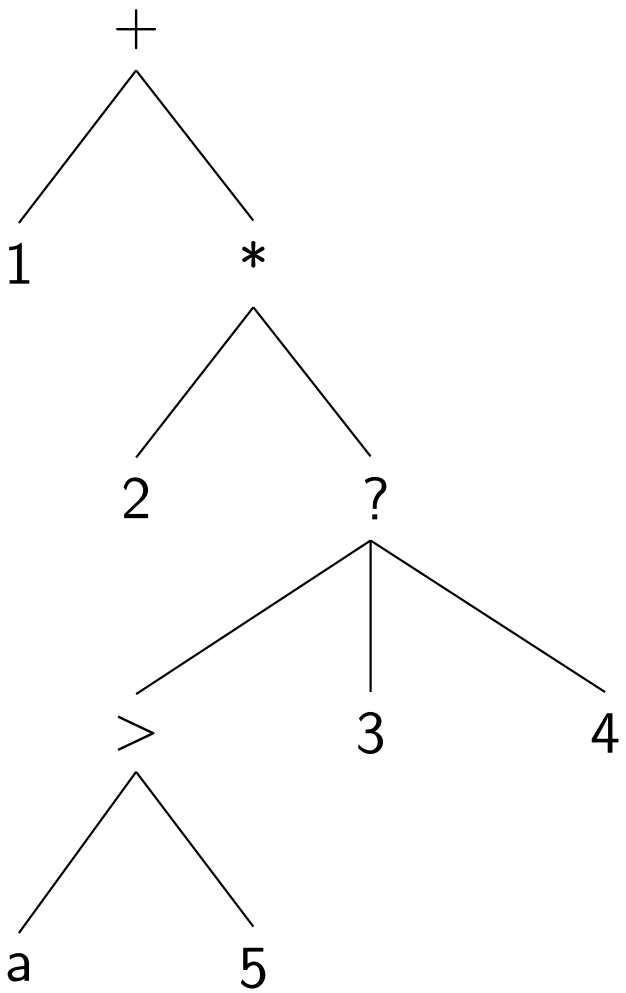
Un arbre dont les nœuds internes sont marqués par :

- des opérateurs tels + ou *
- ou des noms de structures de contrôle ou de structures syntaxiques tels `while` ou block

et dont les feuilles (ou nœuds externes) représentent :

- des opérandes tels un identificateur ou un littéral
- ou des instructions élémentaires telle ' ;' (instruction vide)

Un exemple d'AST d'expression I

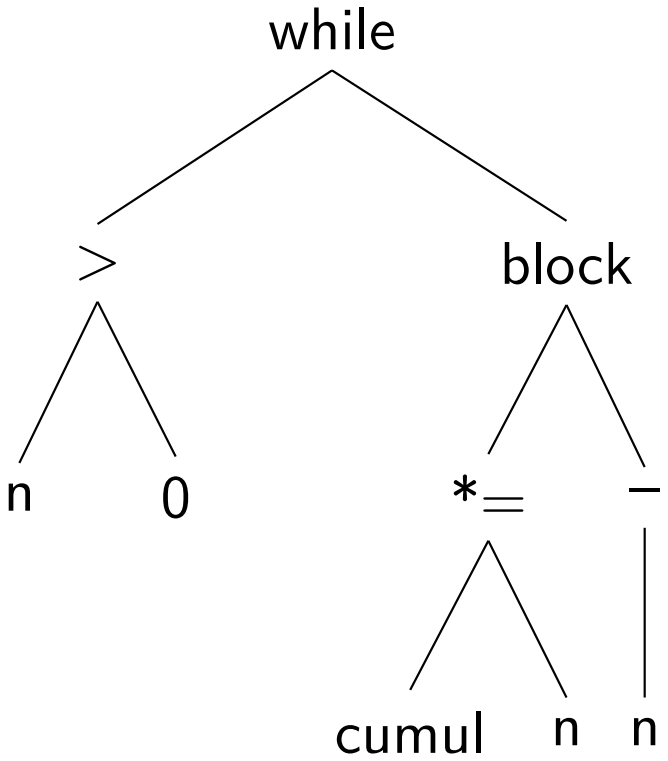


L'AST de gauche représente l'expression entière :

$$1 + 2 * (a > 5?3 : 4)$$

Remarquons que l'AST est une simplification de l'arbre de dérivation (plus de parenthèses ni de symboles non terminaux)

Un exemple d'AST d'itérative I



L'AST de gauche représente l'itération de factorielle :

```
while(n>0){
  cumul*=n;
  n--;
}
```

Là encore, les accolades et des symboles non terminaux tels que instruction ont disparus

Implémentation d'AST I

- l'utilisation d'un langage à objet, tel C++, facilite l'implémentation d'AST
- l'héritage va permettre de spécialiser certains noeuds à partir de la classe de base Noeud
- ainsi la classe Expression permettra de représenter un opérateur et ses n opérandes qui sont des sous-expressions
- la vérification de la compatibilité des types :
  - des sous-expressions (addition de 2 nombres)
  - de chaque sous-expression avec l'opérateur (concaténation nécessite 2 chaînes)
  - l'éventuelle conversion implicite (prévue dans le langage) sera ajoutée dans l'AST (en PHP, 11+"toto" vaut 11)
- l'ajout d'attributs à des noeuds de l'AST est permis durant tout le processus d'analyse (e.g. type de l'expression inféré pendant l'analyse sémantique)

Implémentation d'AST II

- l'AST ainsi attribué est souvent appelé arbre décoré !
- des méthodes génériques peuvent être définies dans la classe de base :
  - dans un compilateur,

    `string genererCode()`

    permettra de générer le code cible (assembleur ou langage intermédiaire) de chaque noeud
  - dans un interpréteur,

    `void executer()`

    pour les instructions permettra de faire évoluer le "contexte d'exécution",

    `Valeur evaluer()`

    pour une expression permettra de calculer la valeur de l'expression

Plan

Tables des symboles I

- Dans les langages à blocs (C/C++, Java, ...), des variables peuvent être définies localement au bloc :

  ```
  for(int i=0; ...){...}
  ```

- chaque bloc d'instruction doit donc être associé à une table des symboles (identificateurs) permettant d'accumuler l'information sur les symboles locaux à ce bloc (nom, type, valeur initiale, adresse ...)
- lors de reconnaissance syntaxique d'un identificateur, la liaison (binding) entre ce dernier et l'entrée dans une table des symboles le représentant peut être complexe voire impossible suivant le langage
- dans la plupart des langages interprétés, la liaison sera dynamique car le symbole sera recherché en remontant les contextes d'exécution et leurs tables des symboles :

Tables des symboles II

```
extern int i; // liaison tardive
int f(int j){ // TDS1
  int c=0;
  for(int i=0; i<10; i++){ // TDS2
    c+=j*i; //
  }
  printf("%d\n", i); // 5
}
```

- dans les langages compilés, la plupart des liaisons sont efféctuées lors de la compilation mais certaines liaisons dites "externes" sont effectuées lors de l'édition de liens
- l'implémentation de chaque table des symboles doit permettre un accès efficace à un symbole par son nom s'il est présent
- dans l'AST, chaque identificateur lié sera représenté par un couple (référence à la TDS, nom ou clef dans cette TDS)

Tables des symboles III

- une implémentation en C++ triviale de TDS consiste à utiliser une table associative ou dictionnaire :

  `std::map<std::string><Symbole>`

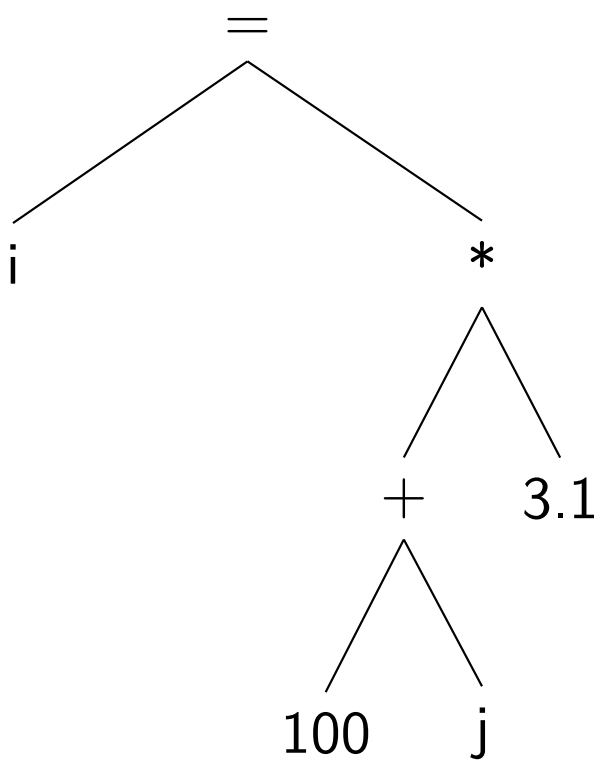
- Symbole agglomérera les différents attribut d'un symbole
- Certains langages, comme le C, ont plusieurs catégories de noms (types, vars, struct, ...) ce qui multiplie encore le nombre de TDS à gérer

Plan

Contrôle de type I

- les langages **statiquement typés** (C/C++, Java, ...) attribuent un type invariable à chaque variable tandis que les langages **dynamiquement typés** (PHP, JavaScript) permettent aux variables d'évoluer également en type (pas seulement en valeur)
- le contrôle de type consiste à vérifier qu'une expression contenant des opérateurs et des opérandes (variables, appels de fonctions, littéraux) puisse être calculée et soit cohérente
- Ce contrôle est réalisé lors de la compilation pour les langages statiquement typés et nécessite parfois des **coercitions de type** (cast) implicites
- dans les langages dynamiquement typés, une partie de ce contrôle pourra être réalisé lors d'une **première passe** mais des coercitions de types seront parfois réalisées lors de l'exécution

Exemple en C I



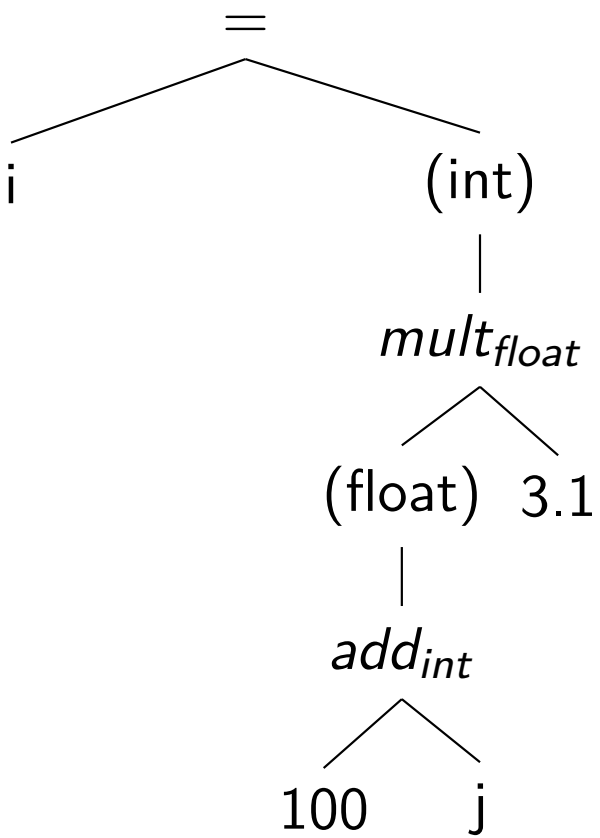
Soit l'expression C

```
i = (100 + j) * 3.1
```

l'analyseur syntaxique a créé l' AST de gauche

en supposant que i, j sont des entiers, il va falloir insérer des cast dans l' **arbre décoré** de la page suivante afin de réaliser les conversions nécessaires

Arbre décoré I



L'addition de 10 avec j est réalisé avec l'addition entière réalisée par l'Unité Arithmétique et logique (ALU)
le résultat est casté en float
la multiplication flottante suivante est effectué dans la FPU
le tout est casté en int afin d'être affecté à i !

Exemple en js I

```
let j=1 ;
let i=(100+j)*3.1 ;
```

- j est intialisé avec le `number` 1.0 (flottant double précision)
- 100 est converti en `100.0`
- l'addition flottante est effectuée (101.0)
- la multiplication flottante puis l'affectation sont réalisées (313.1)
- l'existence d'un unique type numérique simplifie le contrôle de type
- la surcharge d'opérateurs rend le contrôle de type plus complexe

Plan

Calcul de type I

Certains langages utilisent des opérateurs de types afin de construire des types complexes à partir des types primitifs et de ces opérateurs :

```
char * argv[]; // tableau de chaînes
float * pascal[5]; // tableau de ptr (tab) de flottant
struct { struct{ ... } champ1; int champ2[5];} // etc.
```

- il faut définir dans le langage du compilateur ou de l'interpréteur, une structure de données permettant de représenter tous les types possibles
- lors de la compilation du source, il faut construire tous les types utilisés et contrôler la cohérence des types dans les expressions
- là encore, le langage définit parfois des conversions implicites :

```
char t[] = {'H','e','l','l','o','\0'};
char *s = t; // autorisé !
```

Plan

Théorie I

- 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 (entier, 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

Théorie II

- 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

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 :
$G_{ETF} = (\{0,1,\dots,9,+,*,(,)\}, \{E\{val\}, T\{val\}, F\{val\}\}, R, E)$ avec les règles syntaxiques et sémantiques suivantes :

Théorie III

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 Bison I

- Avec Bison, 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

Grammaires attribuées avec Bison II

- 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 Bison implémentant cet exemple :

```
%{                                      /* etf.y  */
#include <stdio.h>                       /* printf */
#include <ctype.h>                       /* isdigit */
#define YYSTYPE int                      /* YYSTYPE comme int */
int yylex(void);void yyerror(char *s);
%}
%token CHIFFRE
%%
liste   :       {/* chaine vide sur fin de fichier Ctrl-D */}
```

Grammaires attribuées avec Bison III

```
        |       liste ligne
        ;
ligne   :       '\n'                {/* ligne vide */}
        |       error '\n'          {yyerrok; /* sync après \n */}
        |       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){ // sans Flex
```

Grammaires attribuées avec Bison IV

```
  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();}
```

Remarques

- en Bison, 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, définitions Bison :

Grammaires attribuées avec Bison V

- `%token<typeDeLUnion> JETON`
- `%type<typeDeLUnion> nonterminal`

Attributs synthétisés I

Définition

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

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 : $x_0.val = f(x_{i1}.a_{i1}, x_{i2}.a_{i2}, \dots, x_{ik}.a_{ik})$.

Attributs synthétisés II

- 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 Bison, 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

Attributs synthétisés III

Définition

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

Les grammaires S-attribuées peuvent facilement être implémenthées avec Bison

Attributs hérités I

Définition

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

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).

Attributs hérités II

- 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

Attributs hérités III

- 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 :

Attributs hérités IV

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 Bison
- 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 Bison
- regardons le contenu de la pile au moment où une production de L est en cours de reconnaissance

Attributs hérités V

- 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 Bison, 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

Attributs hérités VI

- Une implémentation Bison de la grammaire précédente de déclarations est donnée ci-après :

L'analyseur lexical

```
%{                                  /* declar.l */
#define YYSTYPE char *              /* YYSTYPE chaîne */
#include "y.tab.h"                   /* JETONS et yylval */
%}
%option noyywrap
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 ! */}
```

Attributs hérités VII

%%

L'analyseur syntaxique

```
%{                                      /* declar.y */
#include <stdio.h>
#include <string.h>
#define YYSTYPE char *                   /*  YYSTYPE chaine */
int yylex(void);void yyerror(char *s);
int nb; char affich[1024];
%}
%token INT CHAR ID                      /* les jetons (tous chaines) */
%%
inter   :       {/* chaine vide sur fin de fichier Ctrl-D */}
        |       inter {affich[0]='\0';} ligne
        ;
ligne   :       '\n'                    {/* ligne vide : expression vide
↪   */}
```

Attributs hérités VIII

```
        |       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);
        }
```

Attributs hérités IX

```
        |       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)
```

Plan

Méthode de transformation des grammaires L-attribuées I

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

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

- Au moment de réduire par $C \rightarrow c$, 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

Méthode de transformation des grammaires L-attribuées II

- 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

Méthode de transformation des grammaires L-attribuées III

- Ainsi, lorsque la réduction par $C \rightarrow c$ 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 :

Méthode de transformation des grammaires L-attribuées IV

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

Un exemple I

- 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.

Un exemple II

- 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
$S \rightarrow E$	S.val=E.val, E.cal=1	au début, il fa
$E \rightarrow 1$	E.val=1	calcul de base
$E \rightarrow 0$	E.val=0	calcul de base
$E \rightarrow E_1 \|\| E_2$	$E_1.cal = E.cal, E_2.cal = (E.cal?!E_1.val : 0)$ $E.val = (E.cal?(E_1.val?1 : E_2.val) : 99)$	transmission d calcul de l'expr
$E \rightarrow !E_1$	$E_1.cal = E.cal, E.val = (E.cal?!E_1.val : 98)$	calcul de l'expr
$E \rightarrow (E_1)$	$E_1.cal = E.cal, E.val = (E.cal?E_1.val : 97)$	transmission et

Un exemple III

- 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
$S \rightarrow M_1 E$	$S.val = E.val, M_1.cal = 1; E.cal = M_1.val$	au début, il faut calculer
$M_1 \rightarrow \varepsilon$	$M_1.val = M_1.cal$	transmission
$E \rightarrow 1$	E.val=1	calcul de base

Un exemple IV

$E \rightarrow 0$	E.val=0	calcul de base
$E \rightarrow E_1 \|\| M_2 E_2$	$E_1.cal = E.cal, M_2.cal = (E.cal?!E_1.val : 0), E_2.cal = M_2.val$ $E.val = (E.cal?(E_1.val?1 : E_2.val) : 99)$	transmission du court-circuit calcul de l'expression
$M_2 \rightarrow \varepsilon$	$M_2.val = M_2.cal$	transmission du court-circuit
$E \rightarrow !M_3 E_1$	$M_3.cal = E.cal, E_1.cal = M_3.val, E.val = (E.cal?!E_1.val : 98)$	calcul de l'expression
$M_3 \rightarrow \varepsilon$	$M_3.val = M_3.cal$	transmission du court-circuit

Un exemple V

$E \rightarrow (M_4 E_1)$	$M_4.cal = E.cal, E_1.cal = M_4.val, E.val = (E.cal ? E_1.val : 97)$	transmission
$M_4 \rightarrow \varepsilon$	$M_4.val = M_4.cal$	transmission du court-circuit

Un exemple VI

- 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 Bison avec des $i pour représenter les attributs sur la pile

Un exemple VII

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 précé
$E \to (M_4 E_1)$	$\$\$ = (\$2?\$3 : 97)$	transmission
$M_4 \to \varepsilon$	$\$\$ = \$ - 1$	on recopie le marqueur précé

Ce qui donne en Bison :

Un exemple VIII

```
        /* 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);}
        ;
```

Un exemple IX

```
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;}
        ;
%%
```

Un exemple X

```
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 I

Exercice

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

Exercice II

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

Ce qui donne en Bison :

```
        /* 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 */
```

Exercice III

```
        |       liste ligne
        ;
ligne   : '\n'       /* ligne vide : expression vide */
        | error '\n'       {yyerrok; /* après fin de ligne */}
        | m1 exp '\n'      {printf("Résultat : %d\n",$2);}
        ;
m1      :                  {$$=1;      /* $$=vrai */}
        ;
exp     : exp '|' m2 exp   {$$=($0?($1?2:$4):99); /* condensé */}
        | '!' m3 exp       {$$=($0?!$3:98); /* $0 attribut mi */}
        | '(' m4 exp ')'   {$$=($2?$3:97);}
        | '1'              {$$=1;              /* $$=vrai */}
        | '0'              {$$=0;              /* $$=faux */}
        | exp '&' m5 exp   {$$=($0?($1?$4:0):96); /* condensé */}
        ;
m5      :                  {$$=($-2?!$-1:0);}
        ;
```

Exercice IV

```
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 :

Exercice V

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

Plan

Plan

Introduction I

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

- Le langage intermédiaire est souvent soit un langage de machine virtuelle à pile, soit un langage d'arbre représenté par une notation postfixée
- 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

Introduction II

- 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
- Or tous les navigateurs Internet ont un interpète java
- 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

Introduction III

- C'est un langage pour machine abstraite à pile (on voit la filiation avec Java)

Plan

Machine virtuelle à pile I

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 [5].

Plan

Développement d'un compilateur I

- l'étude du langage source est fondamentale mais n'est pas suffisante
- le choix d'un "bon" langage intermédiaire et du langage de développement du compilateur est important
- il est impensable d'écrire un compilateur en langage d'assemblage
- Dans l'environnement Unix, l'écriture en C ou 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 !

Composition de traducteurs I

- Un compilateur peut être représenté par une forme géométrique en T, notée $S_I O$, 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 : $\text{Pascal}_{C++}C$
- 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 $\text{Pascal}_{C++}C$ par $C++_M M$ 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

Composition de traducteurs II

- 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'$_L$M, puis à traduire à la main ce compilateur dans le langage M, on obtient donc L'$_M$M
- 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$_L$N

Composition de traducteurs III

- Si l'on a à sa disposition un compilateur de L sur une autre machine M, $L_M M$, alors on peut très bien obtenir une version du compilateur fonctionnant sur la machine N de la façon suivante :
  1. compiler $L_L N$ gâce à $L_M M$: on obtient $L_M N$ qui est un compilateur
  2. compiler encore une fois $L_L N$ gâce à ce nouveau compilateur $L_M N$: on obtient donc $L_N N$.
- 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

Plan

Plan

Introduction à l'Interprétation I

Dans cette section nous supposons une interprétration du langage :

- A la fin de la phase d'analyse sémantique et éventuellement de génération de code, nous obtenons :
  - soit un fichier de code intermédiaire (fichier .class Java)
  - soit un arbre décoré associé à des tables de symboles
- il convient dès lors d'exécuter ce code grâce au moteur d'exécution

Exemple : la machine virtuelle Java I

Java Virtual Machine (JVM) permet d'exécuter du byte-code Java :

- la JVM est lancé grâce à la commande : `java Toto` qui exécutera le fichier compilé Toto.class
- elle contient un environnement d'exécution (Runtime Environment) composé des librairies Java nécessaires (System, ...)
- plusieurs versions de JVM existent utilisant des architectures d'exécution différentes :
  - simple interpréteur exécutant le byte code
  - Just In Time Compiler, qui compile en langage machine le byte code au lancement puis qui exécute ce code binaire. Cet algorithme est plus efficace pour le code redondant (corps de boucle)
  - dynamic adaptive compilers (DAC) commence par interpréter le byte code et en stocke une version binaire native. Lors d'une seconde passe sur une séquence d'instructions, c'est la version binaire qui est exécutée. Cela nécessite de stocker les deux versions (byte code et binaire natif)

Exemple : la machine virtuelle Java II

  - enfin way-ahead-of-time (WAT) compile le byte code au moment de la compilation javac du source et réalise l'édition de liens avec une librairie au format binaire (langage compilé)
- en plus de l'exécution, la JVM assure le Garbage Collecting (ramasse-miettes) qui permet de désallouer les objets non référencés
- elle gère également le mécanisme d'exception

Plan

Projet : un interprète récursif I

- on souhaite prototyper un interprète d'un langage simple et connu en s'appuyant sur l'arbre décoré associé à un script
- on utilisera une méthode agile utilisant des sprints courts pour obtenir des versions incrémentales
- le langage de programmation se basera sur la syntaxe du C
- le principe d'interprétation consiste à parcourir récursivement l'arbre décoré (séquence d'instructions)
- les ruptures de séquences (itérations, appels de fonctions) nécessitent de conserver une pile des contextes (adrs retour, paramètres, var. locales)

Sprint 1 : AST I

- On souhaite modéliser un arbre de syntaxe abstrait sur un sous-langage basique
- Un seul type `int` est défini
- 4 opérateurs arithmétiques `+,-,*,/`
- l'affectation et les comparaisons
- une instruction `echo <exp>;`
- pas de fonction ni bloc : un script constitué d'une séquence d'instructions

Réalisation du Sprint 1 I

- Analyseur syntaxique sur une première passe et construction de l'AST : `SeqInston * root;`
- Création d'une hiérarchie de classes C++ héritant de la classe abstraite de base `Noeud`
- permettant une séquence d'instructions `SeqInston`
- Chaque instruction (`Inston`) peut être :
  - Une déclaration (`Declaration`) de variable (`int i;`)
  - Une `Expression` suivie d'un ;
  - Une affectation (`Affectation`) d'expression à une variable (`Lvalue`)
  - Une instruction echo
- Une expression peut être simple (`ValInt, Designation`) ou complexe (`Binaire`) utilisant un opérateur infixe
- Une fois l'AST récupéré, ce premier sprint consistera simplement à reconstruire la chaîne de la séquence d'instructions et à l'afficher

Un exemple I

Soit le code source suivant :

```
int i; i=5+3*2;
int __aZ12_36; __aZ12_36=4;
echo __aZ12_36+i;
```

L'analyse de cet exemple donne :

```
Sprint1$ intc exemple.c
int i;
i=(5+(3*2));
int __aZ12_36;
__aZ12_36=4;
echo (__aZ12_36+i);
Sprint1$
```

Hiérarchie de classes I

```
abstract class Noeud { // tout élément de l'AST est un Noeud
  int jeton;
  Noeud *parent;
  virtual string toString();
}
class SeqInston : Noeud {
  list<Inston *> *instons;
  Valeur* exec();
  string toString();
}
abstract class Inston : Noeud {
  virtual Valeur * exec()=0;
  virtual string toString();
}
class Declaration: Inston {
  string *type;
  string *id;
```

Hiérarchie de classes II

```
  Valeur * exec()=0;
  string toString();
}
class Affectation: Inston {
  Lvalue *lvalue;
  Expression * exp;
  ...
}
abstract class Expression : Inston {
  virtual Valeur * calculer()=0;
}
class Binaire : Expression {
  char op; // opérateur +,-,*,/
  Expression *gauche; // sous-exp de gauche
  Expression *droite; // sous-exp de droite
  Valeur * calculer();
}
```

Hiérarchie de classes III

```
class ValInt: public Valeur {
  int valeur;          // attribut spécifique de ValInt
  ValInt(int i):Valeur('I'), valeur(i){}
  int getInt();
  Valeur * calculer(){return new ValInt(valeur);}
```

L'analyseur syntaxique I

```
%%
liste : { // init. : création d'une liste en fond de pile
  root=$$=new SeqInston(); // création d'une séquence vide
}
|  liste inston          { // au moins une inston
  if(erreur){      // pour ne pas ajouter l'error
    erreur=false;
  } else {         // inston normale
    $$=$1->ajouter($2); // ajout d'une inston à la séquence
  }
}
;

inston : error ';'          { // synchro sur prochaine inston
    erreur=true;
    yyerrok;
  }
```

L'analyseur syntaxique II

```
| lvalue '='          exp ';'   {  // affectation vue comme inston
    $$ = new Affectation($1,$3);
  }
| exp ';'    {
    $$=$1;  // une  expression ; est une instruction
  }
| TYPE ID ';' { $$=new Declaration($1,$2);
  }
| MMECHO exp ';' { $$=new Echo($2);
  }
;
lvalue : ID {
  $$=new Lvalue($1); // chemin d'accès m.t[2]
}
;
exp    : exp '+' exp                    {
    $$=new Binaire('+', $1, $3);
```

L'analyseur syntaxique III

```
  }
| exp '-' exp                    {
    $$=new Binaire('-', $1, $3);
  }
| '-' exp          %prec MOINSUNAIRE {
    $$=new Binaire('-', new ValInt(0) , $2);
  }
| exp '*' exp                    {
    $$=new Binaire('*', $1, $3);
  }
| exp '/' exp                    {
    $$=new Binaire('/', $1, $3);
  }
| LITENT   { $$=new ValInt($1);
  }
| ID       { $$=new Designation(*$1); // ID est de type string*
  }
```

L'analyseur syntaxique IV

```
;
%%
int main (int argc, char ** argv, char **env) {
  //yydebug=1;
  if (argc>1){        // $ intc toto.c
      yyin=fopen(argv[1],"r");
      if (yyin==NULL){
          perror("fopen");
          exit(errno);
      }
  }
  int res=yyparse(); // lancement du parser, récup. AST dans la
  ↪  var globale root
  if (res!=0){
    cerr<<"Erreur de syntaxe !"<<endl;
    exit(1);
  }
```

L'analyseur syntaxique V

```
    cout<< "FIN EXEC: ast =" << endl << root->toString(); //
    ↪  affichage de l'arbre
}
```

Plan

Sprint 2 : exécution I

- On conserve le même mini-langage mais on veut pouvoir exécuter le script
- Pour cela, il faut gérer une table des symboles locaux à la séquence d'instruction : structure en bloc `{...}` avec une table des symboles locale à chaque bloc. Plus une table des symboles pour les identificateurs globaux. On n'utilisera qu'un seul espace de nom pour les variables et les fonctions (pas de struct ou étiquettes de goto)
- La liaison entre un identificateur et sa définition sera réalisée lors de l'analyse sémantique.
- Puisqu'il n'y a pas de rupture de séquence, pas besoin de pile de contextes
- l'exécution sera réalisée par parcours récursif de l'AST avec une seule table de symboles globaux

Plan

Sprint 3 : fonctions I

- On crèe le bloc et la fonction dont les identificateurs sont stockés dans la table des symboles globaux
- On ajoute l'appel de fonction qui nécessite un pile de contextes mémorisant l'adrs de retour les paramètres passés et les variables automatiques lié à cet appel
- le compteur ordinal est matérialisé par un pointeur sur l'Instruction suivante
- la structure d'exécution est constituée de cette pile de contexte et du compteur ordinal : pas de structure mémoire (sprint 4 ...)

Plan

Conclusion I

Quelques remarques sur ce cours :

- Théorie des langages formels et programmation efficace
- Découpage entre les phases d'analyse qui n'est pas toujours vérifié dans la pratique (anal. lex., synt. et sém en 1 passe)
- Nombreux outils (ANTLR, JITC, ...)
- Permet de comprendre les concepts fondamentaux des langages (à classes, à prototypes, impératifs, fonctionnels)
- L'efficacité des outils développé influe sur l'orientation des développements professionnels (V8 Javascript Engine qui a propulsé Node et Chrome et les front-end js (Angular, React, Vue))

Bibliographie I


Grune, Bal, Jacobs, Langendoen, Compilateurs, Dunod (2002), col. Sciences Sup., “très technique, beucoup d’algos, 774 pages”


Aho, Sethi, Ullman, Compilateurs, Principes, Techniques et outils, Interéditions (1989), col. IIA, “La bible, indispensable pour la compilation, aspect info., 870 pages”


S. B. Lippman, Le modèle objets du C++, Int. Thomson Pub. (1996), “Très technique, peu pédagogique, les entrailles du C++, 260 pages”


D. Flanagan, Java in a nutshell, O’Reilly Ed., 2nd ed. (1997), “Le livre de référence sur java, 600 pages”


J. Meyer, T. Downing, Java Virtual Machine, O’Reilly Ed. (1997), “Le byte-code de java, 420 pages”

Bibliographie II


A. Ellis, B. Stroustrup, The annotated C++ reference manual, Addison Wesley Ed. (1990), “L’ouvrage de référence sur le C++ et son implémentation, 440 pages”


J. R. Levine, T. Mason, D. Brown, Lex et Yacc, O’Reilly Ed. (1994), “Pour apprendre à programmer en lex et yacc, 330 pages”


Salomaa, Formal Languages, Academic Press (1973), col. ACM monograph series, “Très formel, en anglais, aspect maths, 320 pages”