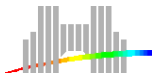
Réaliser un Compilateur
Licence Informatique

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

28 janvier 2009



GENERALITES

A quoi sert un compilateur

- Un ordinateur exécute du code binaire
- Les programmeurs veulent un langage de haut niveau :
  - clarté, mnémotechnie, compacité
  - redondance (permet les vérifications)
  - proximité avec les usages (mathématique, transactionnel, schématique, etc)
- Il faut une traduction, et donc un programme traducteur.

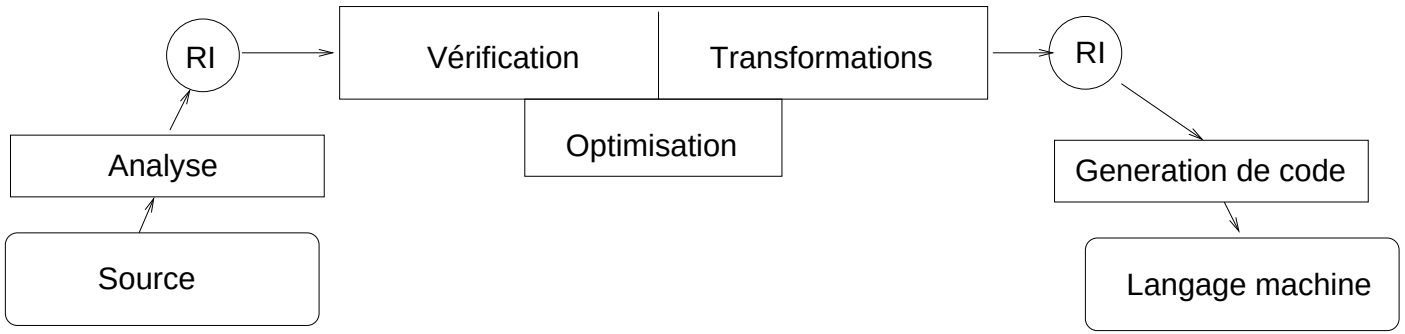
Pourquoi un projet de compilation ?

- C'est un sujet de recherche actif, parce que les processeurs modernes devienenent de plus en plus complexes, et demandent des compilateurs de plus en plus sophistiqués.
- En particulier les processeurs *Dual Core* et multicore posent un problème non résolu : paralléliser un programme arbitraire.
- Les compilateurs les plus utilisés (`gcc`, `javac`, `...`) sont des logiciels libres.
- Beaucoup de problémes se résolvent élégamment en définissant un langage spécialisé et en écrivant son compilateur (exemple : la synthèse matérielle).

Organisation du Projet

- Le but du projet est de réaliser un compilateur naïf pour un langage simple (une réduction de Pascal)
- Prélude au cours de Compilation de M1, excercice de programmation substantiel, travail en commun. Le projet doit être réalisé par binomes
- Quelques cours magistraux (éléments de compilation, présentation du langage Pascal- et de la machine cible), puis travail personnel et séances de d'assistance avec Colin Riba et Bogdan Pasca
- Langage de réalisation recommandé : OCaml
- Evaluation du module : démonstration du compilateur, manuel de l'utilisateur et rapport d'implémentation.

Comment marche un compilateur



- Le code source est analysé suivant une grammaire formelle et codé en binaire (représentation intermédiaire).
- On vérifie que le programme est sémantiquement correct. Exemple : un identificateur est utilisé conformément à sa déclaration.
- On applique des transformation successives qui rapprochent le programme de haut niveau d'un programme en langage machine. Exemples :
  - Boucle $\Rightarrow$ combinaison de tests et de `goto`.
  - Eclatement des expressions complexes : code à un opérateur par instruction.
- Lorsque le code est suffisamment proche du langage machine, on le transcrit en langage d'assemblage.
- Au passage, on essaie d'améliorer le programme (optimisation).

Le langage Ocaml, I

Le langage Ocaml est particulièrement bien adapté pour l'écriture de compilateurs. C'est un langage mixte – fonctionnel et impératif – à typage fort.

- Structure de données récursives :

```
type operon = {op : string;
               args : expression list;
              }
and expression = None
               | Var of string
               | Const of int
               | SubExpression of expression
;;
```

- Ocaml possède un *ramasse-miette* : le programmeur n'a pas à détruire les structures de données devenues inutiles. Pas de fuite de mémoire, pas de pointeurs brisés.

Le langage Ocaml, II

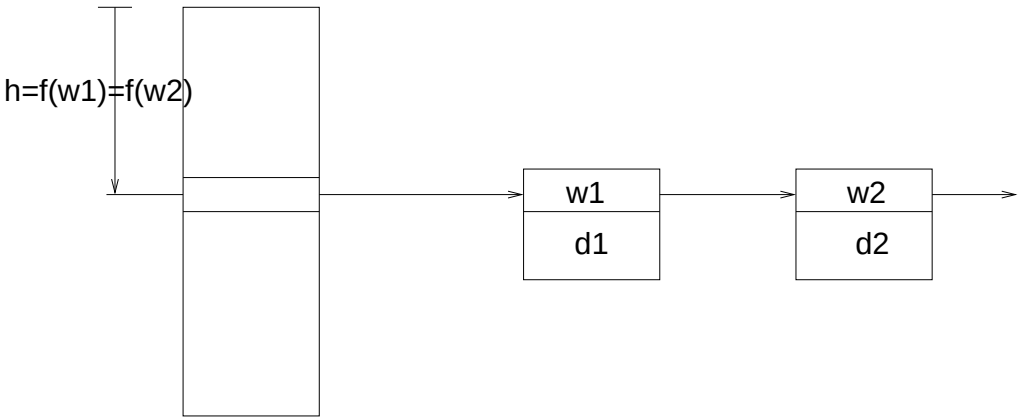
- *pattern matching*

```
match x with
    None -> ...
  | SubExpression e -> ... (List.fst e.args) ...
  | _ -> ()
```

- récursion, listes, chaînes de caractères, tableaux, piles, FIFO, fonctions anonymes, sérialisation, exceptions ...
- objets et foncteurs : utilisation délicate

Le langage Ocaml, III

Tables hashcodées.



Interface :

```
let h = Hashtbl.create 127;
Hashtbl.add h w d;
let d = try Hashtbl.find h w
        with Not_found -> ....
in ...;
```

Langages formels, I

- Un langage est un ensemble de conventions entre un émetteur et un récepteur qui leur permet de se comprendre.
- Emetteur : le programmeur. Récepteur : le compilateur.
- Comme le récepteur n'est qu'un programme, le langage doit être spécifié de fa con précise : langage *formel*.
- On distingue :
  - La *syntaxe* (ou grammaire) : quel sont les programmes corrects.
  - La *sémantique* : quels sont les programmes qui on un sens.
- Analogie avec les langues naturelles.

Langages formels, III

- Un langage formel est un sous ensemble de tous les mots possibles sur un alphabet donné A. Un langage peut être infini.
- Une grammaire est une représentation finie d'un langage.
- Une grammaire formelle se définit à l'aide d'un alphabet auxiliaire N (les non terminaux) et de *règles de réécriture* $a \longrightarrow b$ où a et b sont des mots sur l'alphabet $A \cup N$.
- On passe d'un mot u à un mot v à l'aide de la régle $a \longrightarrow b$ en trouvant dans u une occurence de a et en la rempla cant par b.
- Le langage associé est l'ensemble des mots terminaux engendrés par réécriture à partir d'un non terminal Z (l'axiome).

Langages formels, IV : classification

- $X, Y, Z, \ldots$ sont des symboles terminaux,
- $u, v, w, \ldots$ sont des mots terminaux
- $a, b, c \ldots$ sont des mots arbitraires.
- Grammaires régulières : règles $X \longrightarrow u$ ou $X \longrightarrow Yv$.
- Grammaires algébriques ou hors contexte : règles $X \longrightarrow a$.
- Grammaires sensibles au contexte : règles $aXb \longrightarrow acb$.
- Grammaires générales : pas de restrictions sur les règles.

Seules les deux premières catégories engendrent des langages décidables.

Exemple : la grammaire des identificateurs

$$\begin{array}{llll} \mathcal{I} \to a, & \ldots & \mathcal{I} \to z \\ \mathcal{I} \to A, & \ldots & \mathcal{I} \to Z \\ \mathcal{I} \to \mathcal{I}a, & \ldots & \mathcal{I} \to \mathcal{I}z \\ \mathcal{I} \to \mathcal{I}A, & \ldots & \mathcal{I} \to \mathcal{I}Z \\ \mathcal{I} \to \mathcal{I}0, & \ldots & \mathcal{I} \to \mathcal{I}9 \end{array}$$

C'est une grammaire régulière.

Expressions arithmétiques : une grammaire simplifiée

$$
\begin{aligned}
\mathcal{E} &\rightarrow \mathcal{I} \\
\mathcal{E} &\rightarrow \mathcal{N} \\
\mathcal{E} &\rightarrow (\mathcal{E} + \mathcal{E}) \\
\mathcal{E} &\rightarrow (\mathcal{E} * \mathcal{E})
\end{aligned}
$$

C'est une grammaire hors contexte. Si l'on omet les parenthèses, la grammaire devient ambigüe. Il y a plusieurs suites de réécritures pour engendrer le même mot.

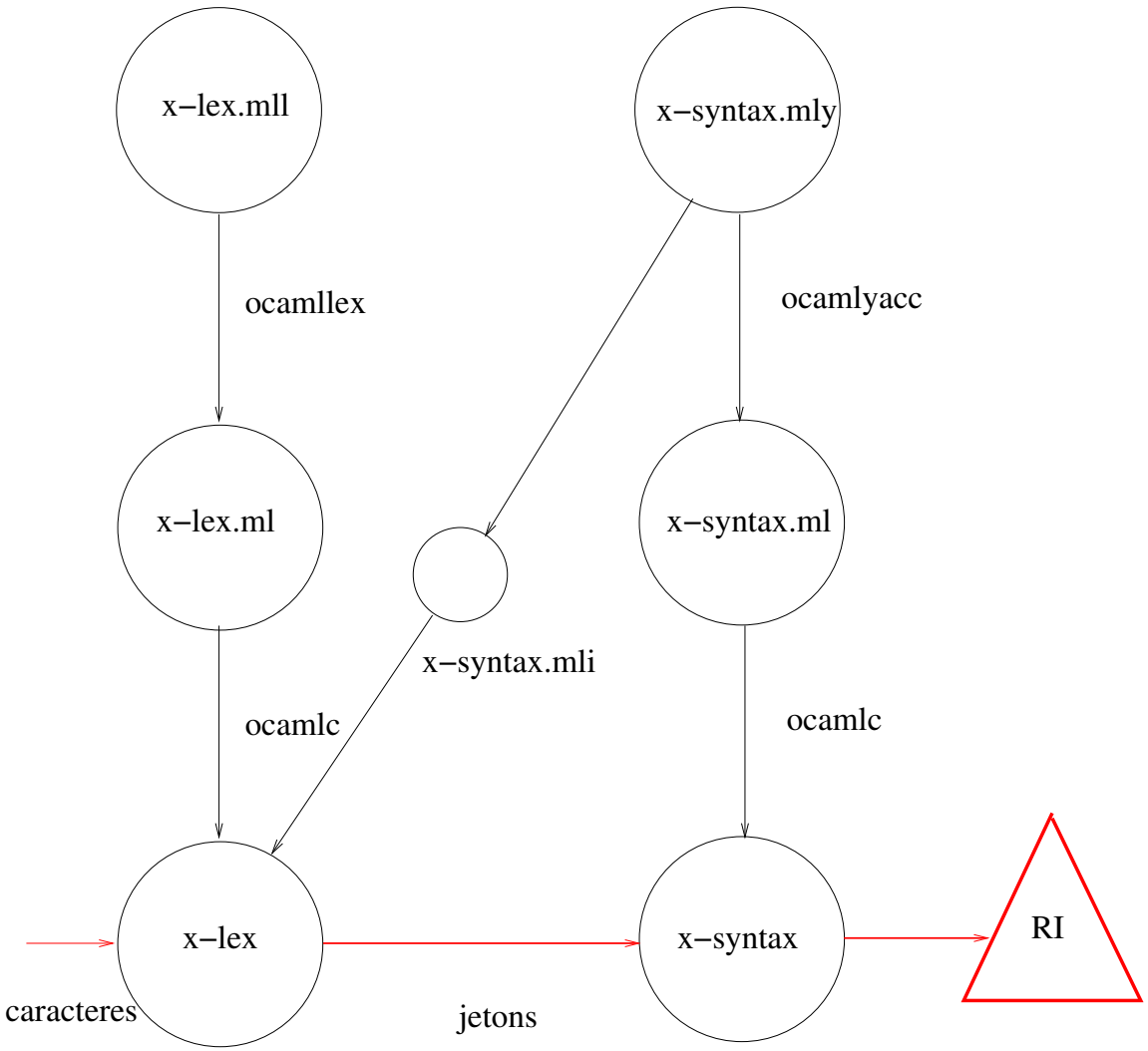
Outillage

- Une grammaire peut être utilisée soit comme générateur, soit comme analyseur.
- Un mot étant donné, reconstituer la suite des productions qui permet de l'engendrer à partir de l'axiome, ou dire qu'il n'est pas dans le langage.
- Ceci n'est possible que pour les deux premiers types de grammaires.
- Pour des raisons d'efficacité, on emploie les deux types, bien que les langages réguliers soient inclus dans les langages hors contexte.
- Mais la puissance des grammaires hors contexte est insuffisante pour exprimer toutes les règles. On doit programmer des contrôles supplémentaires.

Lex et Yacc

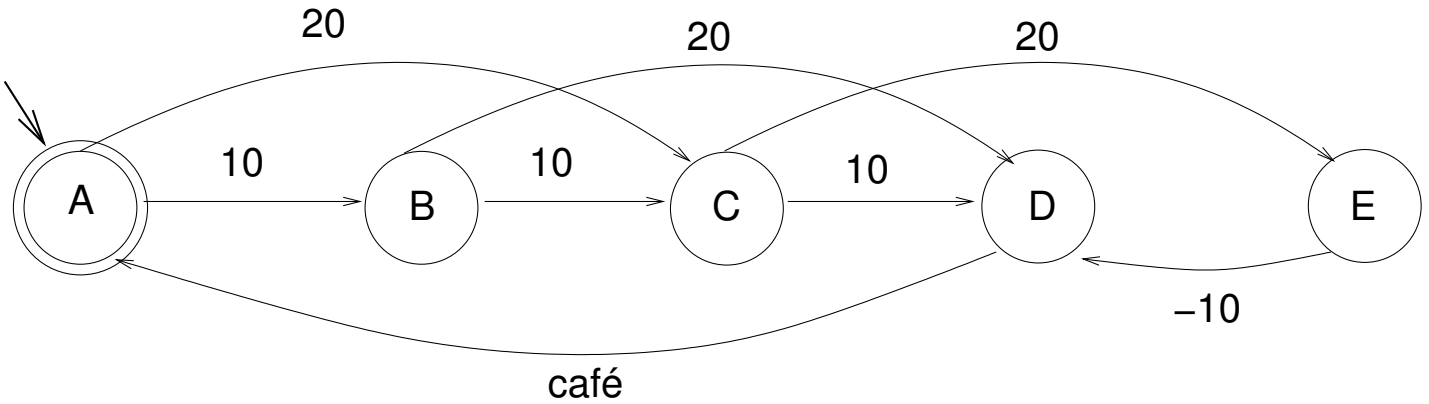
- Analyse lexicale = découpage en *mots* à l'aide de grammaires régulières.
- Analyse syntaxique = découpage en phrases à l'aide d'une grammaire hors contexte.
- On peut inventer des langages formels permettant de décrire les grammaires :
  - liste de productions,
  - conventions lexicales pour distinguer les terminaux, les non terminaux et les opérateurs propres
- On peut écrire des compilateurs pour ces langages : à partir de la grammaire, ils engendrent un analyseur.
- Les deux plus connus sont Lex (pour les grammaires régulières) et Yacc (pour un sous ensemble des grammaires hors contexte).
- Il en existe de nombreuses implémentations qui diffèrent par le langage cible (C, Ocaml, Java, ...) et le statut juridique.

Lex et Yacc, organisation



ANALYSE LEXICALE

Automates finis



L'automate fini de la machine à café.

- Un alphabet d'entrée (ici $\{10, 20\}$) et éventuellement un alphabet de sortie (ici $\{-10, \text{cafe}\}$).
- Un ensemble fini d'états (ici, $\{A, B, C, D, E\}$).
- Un état initial, A qui ici est également terminal.
- Une relation de transition :

$$A.10 \to B \quad A.20 \to C \quad B.10 \to C \quad B.20 \to D$$
$$C.10 \to D \quad C.20 \to E \quad D \to A/\text{cafe} \quad E \to D/-10$$

Définitions

- Un mot est accepté par l'automate s'il existe un chemin étiqueté par les lettres du mot allant de l'état initial à un état terminal.
- L'ensemble des mots acceptés est le langage reconnu par l'automate.
- Le mot de longueur nulle se note ϵ.
- Un automate fini est *déterministe* si la relation de transition est une fonction.
- Un automate peut avoir des ϵ-transitions ; il est alors indéterministe.
- Un état est *accessible* s'il existe un chemin passant par cet état et allant de l'état initial à un état terminal. On peut éliminer les états non accessibles.
- Un automate est complet si pour tout état il y a au moins une transition par lettre de l'alphabet. En général, on complète un automate par des transitions vers un état d'erreur.

Expression régulières

Une représentation textuelle des automates finis. Syntaxe :

$$\mathcal{E} \to \mathcal{L} \quad \mathcal{E} \to \mathcal{E}.\mathcal{E}$$
$$\mathcal{E} \to \mathcal{E}^* \quad \mathcal{E} \to \mathcal{E}|\mathcal{E}$$

- $\mathcal{L}$ est l'ensemble des lettres de l'alphabet.
- Le point représente la concaténation.
- La barre représente l'alternation.
- L'étoile représente l'itération.
- $a.(a+b)^*$ se lit "un a suivi d'un nombre arbitraire de a ou de b" et désigne donc les mots sur l'alphabet $\{a, b\}$ qui commencent par un a.
- Les automates finis et les expressions régulières definissent la même classe de langages.

Identités remarquables

- La concaténation est associative.
- Elle distribue par rapport à l'alternation.
- L'alternation est associative et commutative.
- $X^* = \epsilon | (X.X^*)$.

Lex, flex, ocamllex

- Classification : on se donne une chaîne de caractères et une liste d'expressions rationelles.
- Partitionner la chaîne en "jetons" dont chacun est une instance de l'une des expressions rationelles.
- Il ne doit pas y avoir de portion de chaîne non classifiée.
- Pour chaque jeton, indiquer le "nom" de l'expression rationelle en cause et le texte du jeton (sauf s'il n'y a aucune ambiguité).
- Méthode : construire et minimiser l'automate qui reconnait $(e_1|e_2|\ldots|e_N)^*$ et signaler un jeton chaque fois que l'on passe par l'état terminal.

Expressions régulières augmentées

- Le “programme” soumis à Lex est de la forme :

$$\begin{array}{ll} & e_1\{j_1\} \\ | & e_2\{j_2\} \\ | & \ldots \end{array}$$

- Les expressions rationelles e_i utilisent des raccourcis. Par exemple, `'a'-'z'` représente `'a' | 'b' | 'c' ... | 'z'`.
- Le jeton est décrit par un bout de code dans le langage hote.
- L'automate est construit, déterminisé, minimisé et implémenté sous forme de tables.
- Attention, la taille de l'automate croit très vite avec la longueur des expressions rationelles e_i. Utiliser des astuces (table de mots clef)

Ocamllex : un exemple

```
{
open Parser;;
exception Eof;;
linecount := ref(0);;
}

rule token = parse
    ['\t' ' '] {token lexbuf } (* skip blanks *)
  | ['\n'] {incr linecount; token lexbuf}
  | ['0'-'9']+ as lxm { INT(int_of_string lxm) }
  | '+' { PLUS }
  | '-' { MINUS }
  | '*' { TIMES }
  | '/' { DIV }
  | '(' { LPAREN }
  | ')' { RPAREN }
  | eof { raise Eof }
```

ANALYSE SYNTAXIQUE

Grammaires hors contexte

- Un ensemble $\mathcal{T}$ de symboles *terminaux*.
- Un ensemble $\mathcal{N}$ de symboles *non terminaux*.
- Un ensemble fini de *productions* : $T \rightarrow w, T \in \mathcal{N}, w \in (\mathcal{N} \cup \mathcal{T})^*$.
- Un axiome $S \in \mathcal{N}$.
- Dérivation immédiate : si $T \rightarrow w$ est une production, alors $uTv \rightarrow uwv$.
- Dérivation : $u_0 \rightarrow u_1 \rightarrow \ldots u_n : u_0 \rightarrow^* u_n$.
- Langage engendré : $\{w \in \mathcal{T}^* \mid S \rightarrow^* w\}$.
- Le problème : étant donné un mot et une grammaire, décider s'il peut être engendré et décrire sa construction.

Un exemple : les systèmes de parenthèses

- $\mathcal{T} = \{\underline{(}, \underline{)}, \underline{[}, \underline{]}, \underline{a}\}$.
- $\mathcal{N} = \{S\}$

$$S \to \underline{a} \qquad S \to SS$$
$$S \to \underline{(}S\underline{)} \qquad S \to \underline{[}S\underline{]}$$

$$S \to \underline{[}S\underline{]} \to \underline{[}SS\underline{]} \to \underline{[}a S\underline{]} \to \underline{[}a\underline{(}S\underline{)}\underline{]} \to \underline{[}a\underline{(}a\underline{)}\underline{]}$$

- Langage de Dick, ne peut être engendré par un automate fini.
- Le pouvoir expressif des grammaires hors contexte est supérieur à celui des grammaires rationelles.

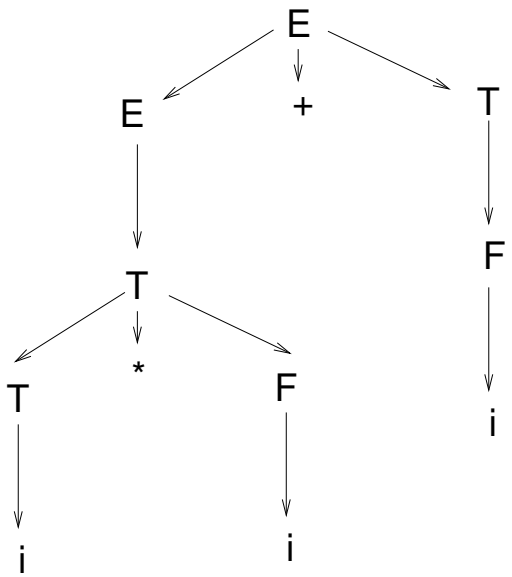
Exemple II : expressions arithmétiques

- $\mathcal{T} = \{i, e, +, -, *, /, (,)\}$
- $\mathcal{N} = \{E, T, F\}$

$$E \to E + T \quad E \to E - T \quad E \to T$$
$$T \to T * F \quad T \to T/F \quad E \to F$$
$$F \to (E) \quad F \to i \quad F \to e$$

$$E \to E+T \to E+F \to E+i \to T+i \to T*F+i \to T*i+i \to i*i+i$$

Arbre d'analyse syntaxique

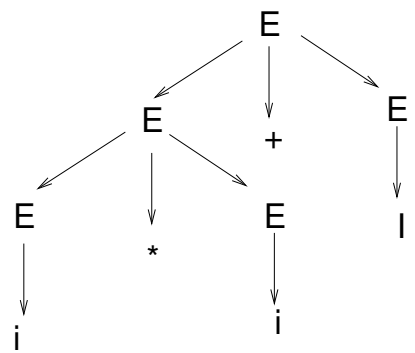


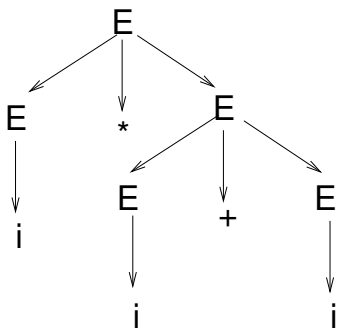
- Les noeuds sont les noms des productions (en général remplacés par le non terminal de gauche)
- Les feuilles sont les terminaux
- Pour chaque application d'une production on crée un noeud étiqueté par le nom de laproduction ayant pour fils les lettres du mot de droite.
- Le mot engendré est la frontière de l'arbre.

Ambiguïté, précédence

Une grammaire est ambiguë s'il existe plusieurs arbres ayant la même frontière.

$$
\begin{array}{rcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & i|e
\end{array}
$$





- On évite les grammaires ambiguës.
- On peut soit compliquer la grammaire, soit inciter l'analyseur à choisir l'une des analyses à l'aide de *règles de précédence*.
- Ici, $* > +$.

Le problème du retour arrière

- L'algorithme d'analyse naïf essaie toutes les productions possibles jusqu'à trouver la bonne : complexité non linéaire.
- Existe-t-il des grammaires pour lesquelles on peut "deviner" la bonne production ?
- Le seule information que l'on possède est le caractère courant de la chaîne d'entrée. Il faut que ce caractère détermine sans ambiguïté la production à utiliser.
- Par exemple, si toute production commence par un terminal, et si pour un non terminal donné, les premiers terminaux de gauche sont distincts, l'analyse syntaxique peut se faire sans retour arrière.

Grammaires prédictives

Pour généraliser, on définit deux fonctions **First** et **Follows** sur l'alphabet des terminaux et non terminaux :

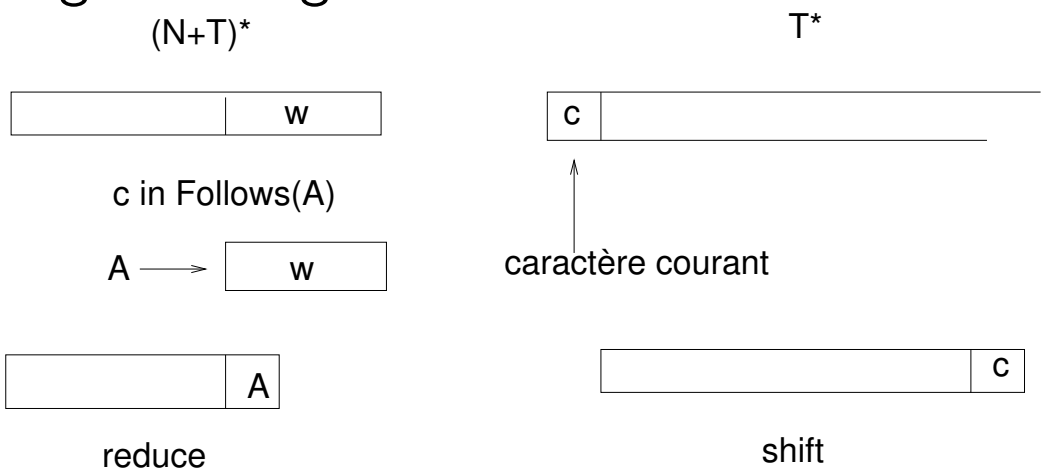
- $x \in \textbf{First}(Y)$ si il existe w tel que $Y \longrightarrow w$ et x est le premier caractère de w. $\epsilon \in \textbf{First}(Y)$ s'il existe une dérivation $Y \longrightarrow \epsilon$.
- En particulier, si y est terminal, $\textbf{First}(y) = \{y\}$.
- $x \in \textbf{Follows}(Y)$ s'il existe une production $A \longrightarrow uYZv$ et $x \in \textbf{First}(Z)$.

$\textbf{First}^+(X) = \textbf{if } \epsilon \in \textbf{First}(X) \textbf{ then } \textbf{First}(X) \cup \textbf{Follows}(X) \textbf{ else } \textbf{First}(X)$;

Une grammaire est prédictive si les membre droits de ses productions ont des ensembles $\textbf{First}^+$ disjoints.

Analyse ascendante

Algorithme glouton



1. On a analysé un préfixe de la chaîne d'entrée.
2. On recherche une production $A \longrightarrow w$ telle que w soit un postfixe de la partie déja analysée et telle que le caractère courant puisse suivre A.
3. Si on trouve, on remplace w par A et on recommence (*reduce*).
4. Sinon, on transporte le caractère courant vers la partie déja analysée, on déplace le pointeur de lecture et on recommence (*shift*).
5. L'algorithme s'arrête quand tous les caractères ont été lu. La chaîne d'entrée est acceptée si la partie analysée se réduit à l'axiome.

Items, fermeture

On veut éviter de parcourir toute la grammaire à l'étape 2. Pour cela, on tient à jour un ensemble de productions candidates ou "items".

- On note que la partie déja analysée est gérée comme une pile.
- On obtient un item en insérant un point dans le second membre d'une production : $A \longrightarrow u \bullet v$.
- Si $A \longrightarrow u \bullet v$ fait partie des productions candidates, cela signifie que u est un postfixe de la pile.
- Si $A \longrightarrow w\bullet$ est une production candidate, on peut réduire.
- Si $A \longrightarrow u \bullet Xv$ est candidate, il faut que l'on puisse pousser un X sur la pile. Toute production de la forme $X \longrightarrow \bullet w$ est donc candidate.
- On construit ainsi la fermeture (*closure*) d'un ensemble d'items.

Automate des items

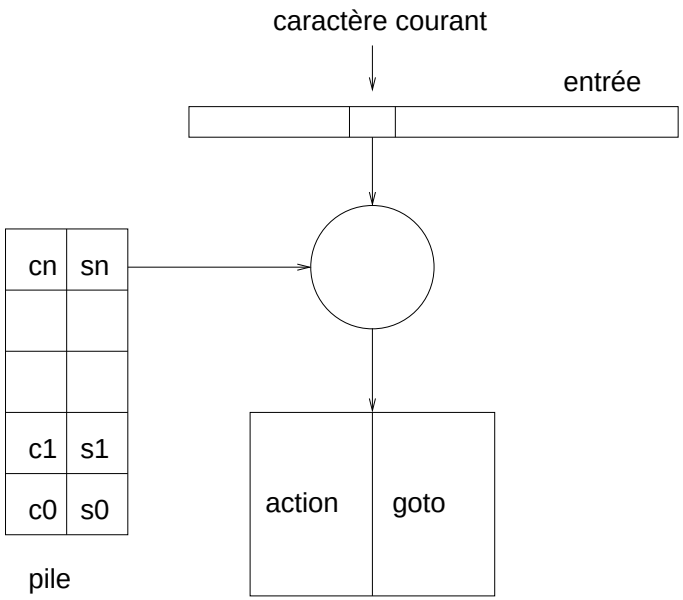
On note que l'ensemble des items est fini, et que l'on peut définir une fois pour toute l'effet de l'empilage d'un caractère quelconque. On construit un automate fini dont les états sont les ensembles d'items.

- Si I est un ensemble d'items, son successeur par le caractére x, terminal ou non terminal, est

$$I.x = \text{closure}(\{A \to ux \bullet v \mid A \to u \bullet xv \in I\})$$

- Pour simplifier, on introduit un nouvel axiome S' et la production $S' \to S$. L'état initial est $\{S' \to \bullet S\}$.
- On utilise un algorithme "à la demande" pour éviter de construire des états inaccessibles.
- On remarque qu'après une réduction on se retrouve dans un état antérieur de l'algorithme, à partir duquel on doit construire le nouvel état courant. Les états doivent donc également être empilés.

Algorithme glouton



Automate à pile

while *true* **do**
  $s = \text{sommet.de.pile};$
  $c = \text{caractere.courant};$
  switch $\text{Action}[s][c]$ **do**
    case *shift*
      push c;
      push $\text{goto}[s][c]$;
    case *reduce* $A \longrightarrow w$
      pop $2 \times |w|$ symboles;
      push A;
      push $\text{goto}[s][A]$;
    case *accept*
      stop;
    case *error*
      error;
      stop;

Ambiguïtés

Lors de la construction de la table Action, on peut rencontrer des ambiguïtés (encore appelées *conflits*) :

- Il peut y avoir dans l'état plusieurs productions réductibles : conflit reduce/reduce.
- On peut avoir le choix entre une réduction $A \longrightarrow w\bullet$ et un shift $B \longrightarrow u \bullet cv$ alors que le caractère courant est c.
- Pour résoudre le problème, on divise les états suivant le caractère courant attendu. Un item prend la forme $[A \longrightarrow u \bullet v, c]$.
- Signification : u est sur la pile, et si on parvient à réduire $u.v$ alors c sera le prochain caractère à lire en entrée.
- On améliore la précision, mais le nombre d'états possibles augmente énormément.

Construction de l'automate LR(1)

- Le calcul de l'automate est identique au cas où on n'utilise pas de symbole *look ahead*.
- Par contre la table des actions est différente :
  - S'il y a plusieurs réduction possibles, on discrimine en fonction du symbole d'entrée suivant. Il n'y a de conflit *reduce / reduce* que s'il y a deux items $[A \longrightarrow w\bullet, a]$ et $[A' \longrightarrow w\bullet, a]$ avec le même terminal a.
  - De même il y a conflit entre une réduction $[A \longrightarrow w\bullet, a]$ et un shift $[B \longrightarrow w \bullet xv, a]$

Résolution des conflits, I

Premier cas : la grammaire est donnée.

- Elle ne doit pas avoir de conflit.
- Si cependant il y en a, l'outil Yacc les résoud de manière arbitraire (en général, priorité au shift plutôt qu'au reduce).
- Il est possible d'agir sur ce choix en spécifiant des priorités ou des associativités.

Le problème du else

```
%{
type statement =
    Assign of string
  | Nop
  | If of string * statement * statement
;;
%}
%token IF THEN ELSE
%token COND
%token ASSIGN
%type <string> COND ASSIGN
%start stat
%type <statement> stat
%%
stat : ASSIGN
       {Assign of $1}
     | IF COND THEN stat
       {If ($2, $4, Nop)}
     | IF COND THEN stat ELSE stat
       {If ($2, $4, $6)}
     ;
%%
```

Le conflit

```
state 7
stat : IF COND THEN . stat  (2)
stat : IF COND THEN . stat ELSE stat  (3)
IF  shift 3
ASSIGN  shift 4
.  error
stat  goto 8

8: shift/reduce conflict (shift 9, reduce 2) on ELSE
state 8
stat : IF COND THEN stat .  (2)
stat : IF COND THEN stat . ELSE stat  (3)
ELSE  shift 9
$end  reduce 2
```

Exemple : une grammaire ambiguë

```
%{
type expression =
   Ident of string
 | Plus of expression * expression
;;
%}

%token IDENT PLUS
%type <string> IDENT
%start expr
%type <expression> expr
%%

expr : IDENT
     {Ident $1}
     | expr PLUS expr
     { Plus $1 $3}
     ;
%%
```

La solution

```
%{
type expression =
   Ident of string
 | Plus of expression * expression
;;
%}

%token IDENT PLUS
%left PLUS
%type <string> IDENT
%start expr
%type <expression> expr
%%

expr : IDENT
     {Ident $1}
     | expr PLUS expr
     { Plus $1 $3}
     ;
%%
```

Résolution des conflits, II

Deuxième cas : le langage est donné, mais pas la grammaire.

- On a probablement construit une grammaire ambigüe.
- L'outil peut afficher l'automate LALR et signale les états qui comportent un conflit.
- Rechercher en particulier les paires de productions qui sont préfixe l'une de l'autre.
- Factoriser.

Résolution des conflits, III

On élabore simultanément le langage et son compilateur.

- On peut ajuster la syntaxe pour simplifier la compilation.
- Prévoir des marqueurs terminaux (par exemple des points-virgules après chaque instruction).
- Utiliser les parenthèses (`begin`, `end`, etc).
- “Refermer” toute les constructions :
  - `if .. then .. else .. fi` (exemple)
  - `do .. end_do`, `proc .. end_proc`

Gestion des erreurs

L'analyseur signale une erreur quand il n'existe aucune transition pour le caractère courant dans l'état courant.

- Le minimum est de signaler l'erreur en indiquant la ligne et le caractère (token) en cause.
- On peut ensuite soit terminer l'analyse (méthode Ocaml) soit essayer de poursuivre (méthode C).
  - Pour poursuivre, on choisit dans le langage une construction facile à repérer (par exemple le point virgule qui termine toute instruction en C).
  - On avale les caractères d'entrée jusqu'à passer un point virgule.
  - On dépile jusqu'à trouver une instruction et on poursuit.

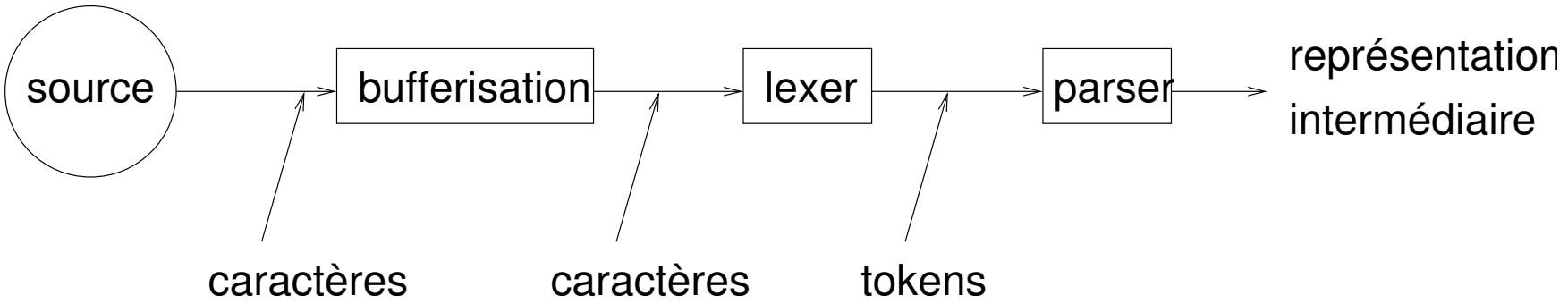
Interface avec Lex, I

- Lex et Yacc doivent utiliser le même système de codage des tokens : techniquement, c'est un type énuméré.
- Dans le contexte Ocaml, c'est ocamlyacc qui construit le type des jetons à partir des déclarations de la grammaire.
- Lex doit importer le fichier d'interface correspondant.

```
type token =
    IF
  | THEN
  | ELSE
  | COND of (string)
  | ASSIGN of (string)

val stat :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> statement
```

Interface avec Lex, II



- L'analyseur lexical et l'analyseur syntaxique fonctionnent en pipeline.
- Techniquement, l'analyseur syntaxique appelle l'analyseur lexical chaque fois qu'il a besoin du jeton suivant.

```
let lexbuf = Lexing.from_channel (open_in fichier)
in try (
    let theSource = Grammar.design Lexicon.token lexbuf
    in ...
  with Parse_error -> ...
```