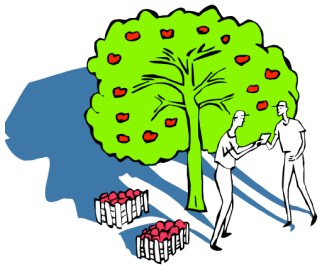
Cours n°13
(euh, presque...)

La syntaxe



Comment passer d'un **texte** à un **arbre** de programme

• Le but de ce cours : écrire en Scheme l'exemple **mfcalc** du manuel **Bison** de GNU. Une calculette interactive !

```
Generating lexer... ok
Generating parser... ok
==> prix = 100.5;
100.5
==> prix-1+2;
101.5
==> prix = 2*prix;
201.0
==> log(prix);
5.303304908059076
==> 1+exp(log(prix));
202.0
==> if prix>100 then 1 else 2;
1
```

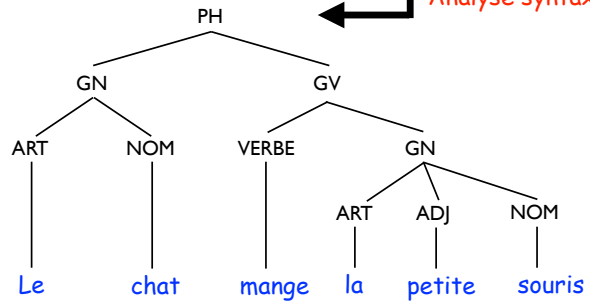
← Génération des analyseurs (lexical et syntaxique)
Lex *Yacc*

- un évaluateur d'expressions infixées,
- avec affectation
- et conditionnelle.

• La compréhension du langage naturel passe par une étape de **structuration** mentale des phrases :

"Le chat mange la petite souris"

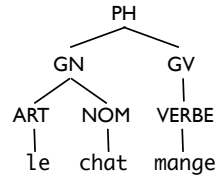
Analyse syntaxique



• Cette analyse syntaxique est précédée par une phase d'**analyse lexicale** : découper le texte en **unités lexicales** ou **lexèmes** (tokens) :

texte ⇒ **Analyseur lexical** ⇒ lexèmes ⇒ **Analyseur syntaxique** ⇒ arbre

"le chat mange" ⇒ "le", "chat", "mange" ⇒



• En réalité, **l'analyseur syntaxique** (*parser*) utilise **l'analyseur lexical** (*lexer*). On ne produit pas tous les lexèmes avant de les analyser !

• Raisonner en termes de **flux de données**...

Les expressions arithmétiques (lexer)

• Lexèmes : *identificateurs* *nombres* + - * / ^ ( )

x	2
prix	456
sin	3.14

```
-x^2 + 454*(y-sin(pi/4))/5 -   10/ x
```

contient 23 lexèmes !

• Nous allons utiliser la bibliothèque **parser-tools** de Racket pour construire un **lexer** capable de générer les lexèmes d'un fichier un par un à la demande… Ensuite viendra le tour du **parser**.

• Le lexer va utiliser des ***expressions régulières*** pour produire un automate fini. Une telle expression re sera :

`identifier` *expands to the named lex abbreviation*

`character` *matches the given character*

`string` *matches the sequence of characters*

`(eof)` *matches end of file*

`(repetition lo hi re)` *matches* re *repeated between* lo *and* hi *times, inclusive.* `hi = +inf.0` *for unbounded repetitions*

`(union re ...)` *matches one of the listed* re *(alternation)*

`(concatenation re ...)` *matches each* re *in succession*

`(char-range char char)` *matches any character between two*

`(complement re)` *matches any character not listed*

`""` *matches the empty string*

• Exemples d'expressions régulières :

digit	(char-range "0" "9")
int	(repetition 1 +inf.0 digit)
int*	(repetition 0 +inf.0 digit)
float	(union (concatenation int "." int*) (concatenation "." int))

```
;;; mfcalc-lexer.rkt
(require parser-tools/lex)
(provide get-lexeme value-tokens op-tokens)   ; pour le parser…

(define-lex-abbrevs
  [whitespace (union #\newline #\return #\tab #\space)]
  [letter (union (char-range "a" "z") (char-range "A" "Z"))]
  [digit (char-range "0" "9")]
  [int (repetition 1 +inf.0 digit)]
  [float (union
          (concatenation int "." (repetition 0 +inf.0 digit))
          (concatenation "." int))]
  etc.)
```


Lexème ≈ token

• Un **lexème** est essentiellement une string.

• Un **token** est un objet plus symbolique, représenté en Racket par une structure de type token, ayant deux champs name et value :

```
#(struct:token VAR prix)   → token-name = VAR
#(struct:token FNCT log)     token-value = prix
#(struct:token NUM 23)
#(struct:token NUM 101.5)  → token-name = NUM
                             token-value = 101.5
```

```
(define-tokens value-tokens (NUM VAR FNCT))

(define-empty-tokens op-tokens (LPAR RPAR + - * / ^ = > < >=
  <= == != EOF SEMICOLON NEG log exp if then else))
```

• Les **empty tokens** ont un champ value vide : #f

• Exemple de fichier `test-lex.dat` à analyser :

```
prix = 2*(
        100.5-1);
if (prix > 100.)==true then .1 else exp(log(prix));
```

automatiquement munies de valeurs dans le lexer...

```
(define get-lexeme
  (lexer
   [(eof) 'EOF]
   [(repetition 1 +inf.0 whitespace) (get-lexeme input-port)]
   [";" 'SEMICOLON]
   ["(" 'LPAR]
   [")" 'RPAR]
   [(union "+" "*" "-" "/" "^" "=" "<" ">" "<=" ">=" "==" "!="
      "if" "then" "else") (string->symbol lexeme)]
   [function (token-FNCT (string->symbol lexeme))]
   [boolean (token-BOOL (string->symbol lexeme))]
   [var (token-VAR (string->symbol lexeme))]
   [(union int float) (token-NUM (string->number lexeme))]))
```


• Le lexer a produit une fonction `(get-lexeme port-in)` qui consomme un port d'entrée pour en extraire les lexèmes :

```
(call-with-input-file "test-lex.dat"
  (lambda (p-in)
    (do ((i 0 (+ i 1)) (x (get-lexeme p-in) (get-lexeme p-in)))
      ((equal? x 'EOF) (list i 'lexèmes 'ont 'été 'lus))
      (printf "----> ~a\n" x))))
```

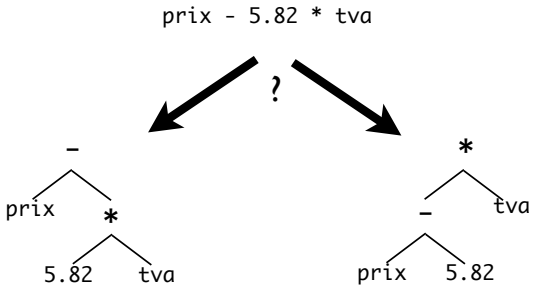
```
----> #(struct:token VAR prix)
----> =
----> #(struct:token NUM 2)
----> *
----> LPAR
----> #(struct:token NUM 100.5)
  ⋮
----> RPAR
----> RPAR
----> SEMICOLON
(29 lexèmes ont été lus)
```


• Si l'on analysait un véritable langage de programmation, on placerait les primitives du langage [`let`, `while`, ...] parmi les **empty tokens**.
• Une chaîne de caractères peut être vue comme un port d'entrée ou de sortie, par exemple pour les tests interactifs :

```
> (let* ((str (begin (printf "==> ") (read-line)))
         (p-in (open-input-string str)))
    (do ((x (get-lexeme p-in) (get-lexeme p-in))
         (s 0 (+ s 1)))
      ((equal? x 'EOF) (list s 'lexèmes))))
==> 12*(5.002-x)
(7 lexèmes)
```

• Maintenant, il reste l'essentiel : organiser les lexèmes en utilisant une grammaire. C'est l'affaire du **parser**...

Les expressions arithmétiques (parser)



(en Java)

• Donc plusieurs arbres syntaxiques possibles.
• Quelle grammaire pour les expressions arithmétiques ?

• La grammaire suivante serait ***ambigüe*** :

E ⟶ E + E	E ⟶ E - E	E ⟶ E * E	E ⟶ E / E
E ⟶ (E)		E ⟶ int	

• L'expression 1+2*3 admet en effet plusieurs dérivations (gauches) possibles :

E ⟶ E*E ⟶ E+E*E ⟶ 1+E*E ⟶ 1+2*E ⟶ 1+2*3

E ⟶ E+E ⟶ 1+E ⟶ 1+E*E ⟶ 1+2*E ⟶ 1+2*3

correspondant à deux arbres syntaxiquement et sémantiquement distincts :

• Deux stratégies s'offrent au programmeur :
- soit *modifier la grammaire* pour la rendre non ambigüe.
- soit *préciser des priorités* pour les opérateurs et laisser le logiciel d'analyse syntaxique [le **parser**] se débrouiller...

• Deux exemples de grammaires non ambigües [ETF] :

G1

E ⟶ T + E	E ⟶ T - E	E ⟶ T
T ⟶ F * T	T ⟶ F / T	T ⟶ F
F ⟶ (E)	F ⟶ int	

récursivité droite

G2

E ⟶ E + T	E ⟶ E - T	E ⟶ T
T ⟶ T * F	T ⟶ T / F	T ⟶ F
F ⟶ (E)	F ⟶ int	

récursivité gauche

• G1 est propice à une **analyse récursive descendante** :

top-down parsing

procedure E :
- appeler la procédure T
- si le prochain lexème est "+" ou "-" :
  alors appeler récursivement la procédure E
  sinon fini, on vient de lire une expression.

procedure T :
- appeler la procédure F
- si le prochain lexème est "*" ou "/" :
  alors appeler récursivement la procédure T
  sinon fini, on vient de lire un terme.

procedure F :
- si le prochain lexème est un entier :
  alors fini, on vient de lire un facteur
  sinon si c'est "(" :
    alors appeler la procédure E;
      puis vérifier que le prochain lexème est bien ")"
    sinon ERREUR

• Cette démarche manuelle **top-down** peut en fait s'obtenir à partir de la grammaire ambigüe en la mettant si possible sous la forme :

LL(1)

Left to right — 1 token de look-ahead

Leftmost derivation

A chaque étape, on développe le non-terminal le plus à gauche

On analyse le texte-source de la gauche vers la droite.

On ne s'autorise qu'à consulter 1 token dans le texte-source

Top-down : on construit l'arbre de dérivation en partant de la racine !

• Il existe des analyseurs généraux LL(1), par exemple en Scheme SLLGEN de Mitchell Wand. C'est l'approche EOPL [Essentials Of Programming Languages, MIT Press].

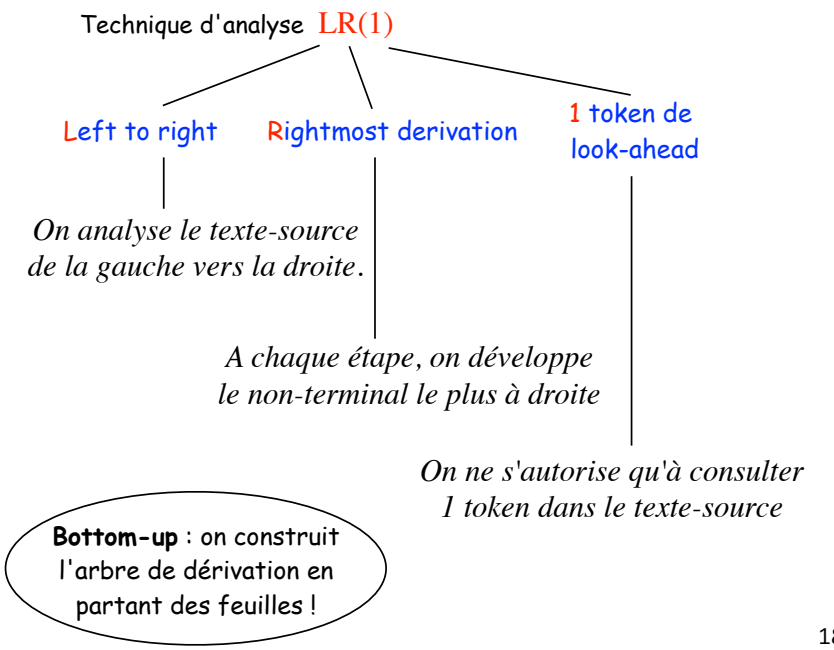
• Mais ce n'est pas la voie que nous suivrons car :
- les grammaires LL(1) peuvent être délicates à produire.
- l'outil le plus utilisé est plutôt :

YACC

Yet Another Compiler Compiler

disponible sous des formes assez proches en C, Caml, Java, Python, Scheme, etc. [chez GNU : Lex=Flex et Yacc=Bison].

• Mais cette fois l'analyse sera **bottom-up** : LR(1)...



• Prenons un exemple de grammaire simple [ambigüe] :

E ⟶ num E ⟶ E * E E ⟶ E + E E ⟶ (E)

```
;;; fichier mfcalc1.rkt - un premier essai !
(require parser-tools/yacc)
(require "mfcalc-lexer.rkt")          ; l'analyseur lexical

(define parse
  (parser
   (start S)                          ; l'axiome de départ
   (end SEMICOLON)                    ; fin d'une analyse sur ";"
   (tokens value-tokens op-tokens)    ; en provenance du lexer
   (error (lambda (a b c) (printf "Error with token ~a\n" b)))
   (grammar
     (S    [() 'EOF]
           [(expr) $1])
     (expr [(NUM) $1]
           [(expr * expr) (* $1 $3)]
           [(expr + expr) (+ $1 $3)]
           [(LPAR expr RPAR) $2])))))
```

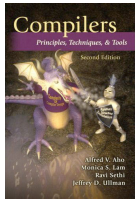
le résultat de l'analyse de expr * expr

• Compilation du parser ==> 4 **shift/reduce conflicts**
• Des *conflits* ? Hum. On essaye quand même ?...

```
(define lex (lambda () (get-lexeme (current-input-port))))

(define (parse-one-expr)
  (printf "==> ")
  (let ((result (parse lex)))
    (if (not (equal? result 'eof))
        (begin (printf "~a\n" result)
               (parse-one-expr)))))

> (parse-one-expr)
==> 2+3+4;
9
==> 2+3*4;
14
==> 2*3+4;
14
==> ;
>
```

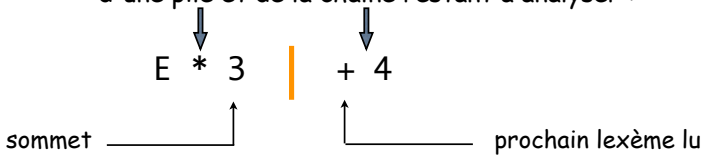
Oups !

• La compilation du texte du *parser* a produit un **automate à pile** qui procède à une analyse **LR(1)**, avec deux actions :
- **shift** : consommer un lexème et l'empiler.
- **reduce** : réduire une production sur le sommet de pile.



• *Détails :*
Compilers. Principles, techniques and tools
Aho & al., Addison-Wesley
+ cours d'analyse au semestre 2.

• Sujet complexe, nous n'entrerons pas dans les détails. La situation générale est constituée :
d'une pile et de la chaîne restant à analyser :



$E \xrightarrow{1}$ num $E \xrightarrow{2} E * E$ $E \xrightarrow{3} E + E$ $E \xrightarrow{4} (E)$

situation initiale :		2 * 3 + 4
shift :	2	* 3 + 4
reduce 1 :	E	* 3 + 4
shift :	E *	3 + 4
shift :	E * 3	+ 4
reduce 1 :	E * E	+ 4

Ah ! Conflit : ***shift*** ou ***reduce 2*** ?

• **Lors d'un conflit shift/reduce, le parser optera pour shift, s'il ne dispose d'aucune autre information** !

shift :	E * E +	4
shift :	E * E + 4	
reduce 1 :	E * E + E	
reduce 3 :	E * E	
reduce 2 :	(E)	

Accepté !

• La **table de transition de l'automate** peut s'obtenir dans un fichier :

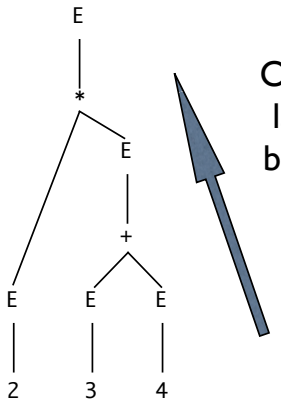
```
(if (file-exists? "mfcalc1-table.txt")
    (delete-file "mfcalc1-table.txt"))

(define parse
  (parser
   (debug "mfcalc1-table.txt")
   (start S)
   (end SEMICOLON)
   ...
)
```

• Mais elle est difficile à exploiter, il est nécessaire de lire le livre cité. Pour le semestre 2 ?...

• Par contre, il est immédiat de tracer l'application des règles.
Par exemple, pour expr + expr, remplacer (+ $1 $3) par :

```
(begin (printf "règle3: E=E+E=~a+~a\n" $1 $3) (+ $1 $3))
```

```
==> 2*3+4;
règle1: E=num=2
règle1: E=num=3
règle1: E=num=4
règle3: E=E+E=3+4
règle2: E=E*E=2*7
14
```



On voit bien la stratégie bottom-up !

• Il reste que l'arbre de dérivation est **INCORRECT** !

• Le conflit provient bien entendu de l'**ambiguité** de la grammaire. On peut lever ce conflit en déclarant des **priorités** et des **associativités** d'opérateurs.

```
-2^3+4*5 = (-(2^3))+(4*5)
```

```
2^3^4 = 2^(3^4)
      ≠ (2^3)^4
```

^ est *associatif à droite*

```
2-3-4 = (2-3)-4
      ≠ 2-(3-4)
```

- est *associatif à gauche*


```
;;; fichier mfcalc2.rkt - correct mais incomplet !
(define parse
  (parser
   (start S)
   ...
   (precs (left - +)
          (left * /)
          (left NEG)
          (right ^))
   (grammar
    (S    [() 'eof]
          [(expr) $1])
    (expr [(num) $1]
          [(expr * expr) (* $1 $3)]
          [(expr + expr) (+ $1 $3)]
          [(expr - expr) (- $1 $3)]
          [(expr / expr) (/ $1 $3)]
          [(lpar expr rpar) $2]
          [(- expr) (prec NEG) (- $2)]
          [(expr ^ expr) (expt $1 $3)])))))
```

associativités

priorités croissantes

```
==> -2^3+4*5;
12
==> -(2+3.5)*2-1;
-12.0
==> 2^3^4;
241785163922925834941235 2
==> (2^3)^4;;
4096
>
```


• Introduisons la conditionnelle `if ... then ... [else ...]`

```
E ⟶ if E then E else E
E ⟶ if E then E
E ⟶ autres
```

• Cette grammaire est ambigüe [*"dangling else" problem*] :

```
if p then if q then r else s
```

?

```
(if p (if q r s))     ou     (if p (if q r) s)  ?
```

• C'est encore un ***conflit shift/reduce*** [et le shift gagne !] :

```
if p then if q then r | else s
```

• On peut laisser le conflit et le shift gagner tout seul. Un else sera alors associé au if le plus proche... Hum.

• On peut supprimer le conflit en réglant les priorités :

```
(precs (nonassoc fictif)        ; opérateur fictif
       (nonassoc else))         ; moins prioritaire que else

[(if expr then expr else expr) (if $2 $4 $6)]
[(if expr then expr) (prec fictif) (if $2 $4 (void))]))))
```

```
if p then if q then r | else s
```

else est prioritaire sur la règle à réduire

• On peut modifier la grammaire pour supprimer le conflit [cf cours d'Analyse].

• On peut rajouter un *end-if* fermant, comme en Maple :

```
if x>2 then x:=x+1 fi
```


• Les **conflits reduce/reduce** sont plus rares et dénotent souvent des erreurs de conception dans la grammaire. Ex:

E ⟶ foo	E ⟶ bar	foo ⟶ num	bar ⟶ num

```
(expr [(foo) $1]
      [(bar) $1])
```

fichier "red-red.scm"

```
(foo  [(num) (begin (printf "Règle foo\n") $1)])
(bar  [(num) (begin (printf "Règle bar\n") $1)])
```

1 reduce/reduce conflict

```
? 123;
Règle foo
==> 123
```

shift : 123

reduce :

123

On réduit avec foo ou bar ?

• En cas de conflit reduce/reduce, la règle déclarée en premier est appliquée !...

La gestion des variables

• Il reste dans notre `mfcalc` à gérer les variables. Nous maintenons leurs valeurs dans une **table de hash-code**.

```
(define vars (make-hash-table))
```

ou "adressage dispersé"

• On accèdera à la valeur de v par `(hash-table-get vars v)` et on traduira l'affectation v=val par `(hash-table-put! vars v val)`.

• Les *précédences* :

```
(precs (nonassoc < > <= >= == !=)
       (nonassoc LOWER_THAN_ELSE)
       (nonassoc else)
       (right =)
       (left - +)
       (left * /)
       (left NEG)
       (right ^))
```


fichier "mfcalc4.scm"

```
(expr [(num) $1]
      [(bool) (eq? $1 'true)]
      [(var) (hash-table-get vars $1 (lambda () 0))]
      [(var = expr) (begin (hash-table-put! vars $1 $3) $3)]
      [(fnct lpar expr rpar) ((eval $1) $3)]
      [(expr + expr) (+ $1 $3)]
      [(expr - expr) (- $1 $3)]
      [(expr * expr) (* $1 $3)]
      [(expr / expr) (/ $1 $3)]
      [(- expr) (prec NEG) (- $2)]
      [(expr ^ expr) (expt $1 $3)]
      [(expr > expr) (> $1 $3)]
      [(expr >= expr) (>= $1 $3)]
      [(expr < expr) (< $1 $3)]
      [(expr <= expr) (<= $1 $3)]
      [(expr == expr) (equal? $1 $3)]
      [(expr != expr) (not (equal? $1 $3))]
      [(lpar expr rpar) $2]
      [(if expr then expr else expr) (if $2 $4 $6)]
      [(if expr then expr) (prec LOWER_THAN_ELSE)
                           (if $2 $4 (void))])
```

0 conflict !

• Deux remarques :

- la stratégie bottom-up préfère les **RÉCURSIVITÉS GAUCHES** alors que la stratégie top-down les déteste !

- si dans une règle, on remplace un calcul par de la construction de code [Scheme par exemple], on obtient un traducteur de syntaxe (cf TP) :

```
[(expr + expr) (+ $1 $3)]

[(expr + expr) (list '+ $1 $3)]

[(expr + expr) `(+ ,$1 ,$3)]
```

• Maintenant c'est vraiment fini. Ouf !