

Algorithmes, Programmation, IA

Cours 6

Jean-Jacques Lévy

`jean-jacques.levy@inria.fr`

<http://jeanjacqueslevy.net/algo-prog-ia-25>

Plan

- fonctions sur les chaînes de caractères
- analyse lexicale
- analyse syntaxique
- évaluation d'expressions arithmétiques

dès maintenant: **télécharger Python 3 en** `http://www.python.org`

Quelques rappels

- le cours utilise le langage Python et l'environnement Visual Studio Code. (`vscode`)
- et pour la partie IA, la bibliothèque Pytorch

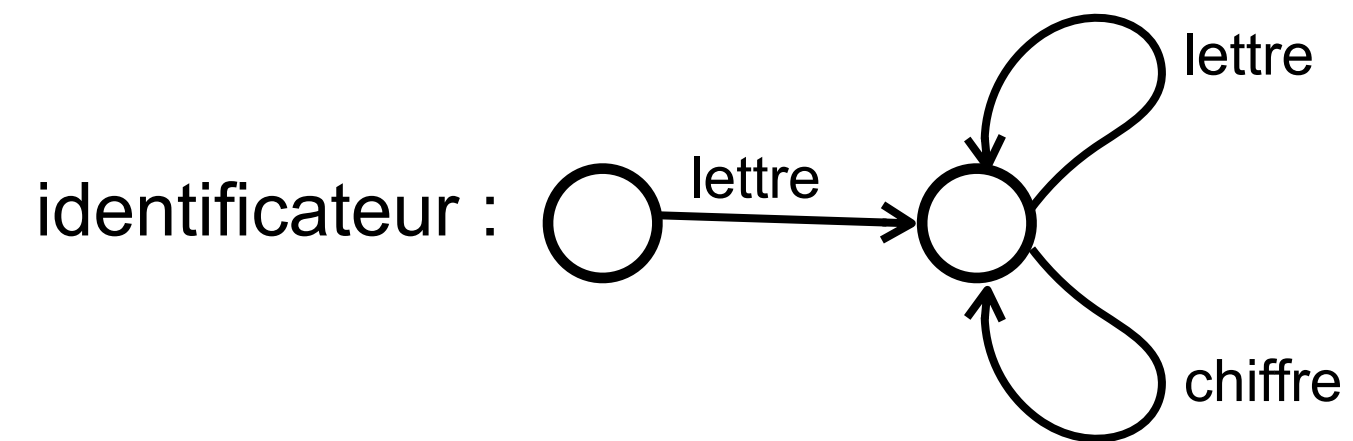
Analyse lexicale

- isoler les mots importants (*lexèmes*) dans une chaîne de caractères (ou fichier de texte)
- suppression des espaces, tabulations, retours ligne

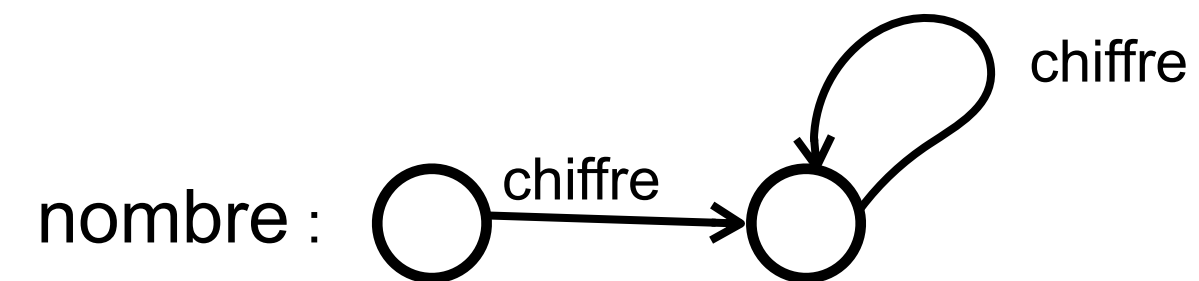
```
txt = '(5 + 2 * 3) * (10 * 10 + 9 * 9)'  
print(txt.split())
```

➡ ['(', '5', '+', '2', '*', '3)', '*', '(', '10', '*', '10', '+', '9', '*', '9)']

- insuffisant car pas de distinction entre *lexèmes* non séparés par des espaces
- automates finis pour les distinguer



opérateur : { '+', '*', '-', '/' }



parenthèse : { '(', ')' }

Analyse lexicale

- isoler les mots importants (*lexèmes*) dans une chaîne de caractères (ou fichier de texte)
- suppression des espaces, tabulations, retours ligne

`txt = '(5 + 2 * 3) * (10 * 10 + 9 * 9)'`  `[ '(', '5', '+', '2', '*', '3', ')', '*', '(', '10', '*', '10', '+', '9', '*', '9', ')' ]`

- description aussi possible avec expressions régulières

<i>lettre</i>	=	$a b \dots z A B \dots Z _$
<i>chiffre</i>	=	$0 1 2 3 4 5 6 7 8 9$
<i>identificateur</i>	=	$lettre(lettre chiffre)^*$
<i>nombre</i>	=	$chiffre^+$

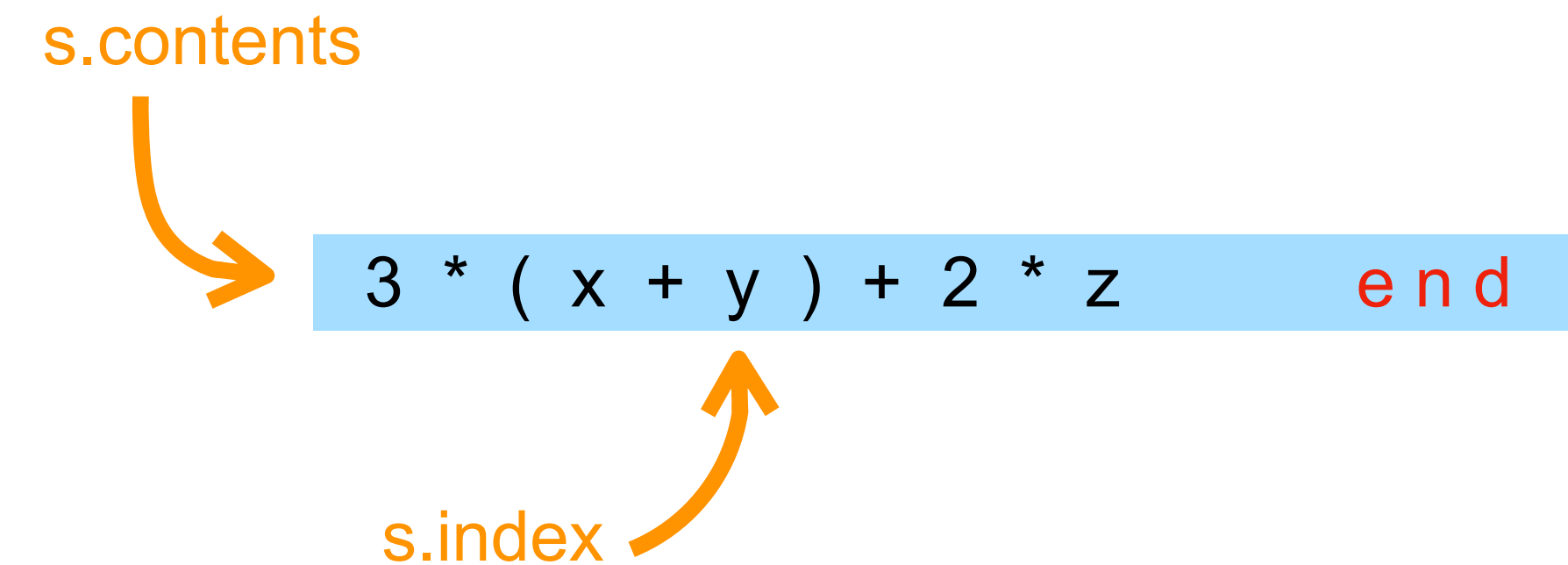
Analyse lexicale

- isoler les mots importants (*lexèmes*) dans une chaîne de caractères (ou fichier de texte)
- suppression des espaces, tabulations, retours ligne

```
lettre = {chr(i) for i in range(ord('a'), ord('z') + 1)}  
chiffre = {chr(i) for i in range(ord('0'), ord('9') + 1)}  
op = {'+', '*', '-', '/'}  
blancs = {' ', '\t', '\n'}
```

```
class Stream :  
    def __init__ (self, txt) :  
        self.contents = txt + ' end ' ← marqueur de fin  
        self.index = 0  
    def progress (self) :  
        self.index += 1  
    def current (self) :  
        return self.contents[self.index]
```

```
def skip_whites (s) :  
    while s.current() in blancs :  
        s.progress()
```



Analyse lexicale

- isoler les mots importants (*lexèmes*) dans une chaîne de caractères (ou fichier de texte)
- suppression des espaces, tabulations, retours ligne

```
def next_lexem (s) :  
    skip_whites (s)  
    c = s.current()  
    if c in lettre:  
        return get_ident(s)  
    elif c in chiffre:  
        return get_number(s)  
    elif c in op :  
        s.progress()  
        return ('OP', c)  
    elif c == '(' :  
        s.progress()  
        return ('PAR0', None)  
    elif c == ')' :  
        s.progress()  
        return ('PAR1', None)  
    else: raise Exception ("Caractère illégal")
```

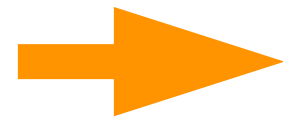
```
def get_ident(s):  
    r = ''  
    while s.current() in union(lettre, chiffre) :  
        r = r + s.current()  
        s.progress()  
    return ('ID', r)
```

```
def get_number(s):  
    r = 0  
    while s.current() in chiffre:  
        r = r * 10 + ord(s.current()) - ord('0')  
        s.progress()  
    return ('INT', r)
```

Analyse lexicale

- isoler les mots importants (*lexèmes*) dans une chaîne de caractères (ou fichier de texte)
- suppression des espaces, tabulations, retours ligne

```
txt = '3 * (x + y) + 2 * z'
s = Stream(txt)
print("texte = '%s' " %txt)
for i in range(12):
    print(next_lexem(s))
```



```
texte = '3 * (x + y) + 2 * z'
('INT', 3)
('OP', '*')
('PAR0', None)
('ID', 'x')
('OP', '+')
('ID', 'y')
('PAR1', None)
('OP', '+')
('INT', 2)
('OP', '*')
('ID', 'z')
('ID', 'end')
```


Arbres de syntaxe abstraite (ASA)

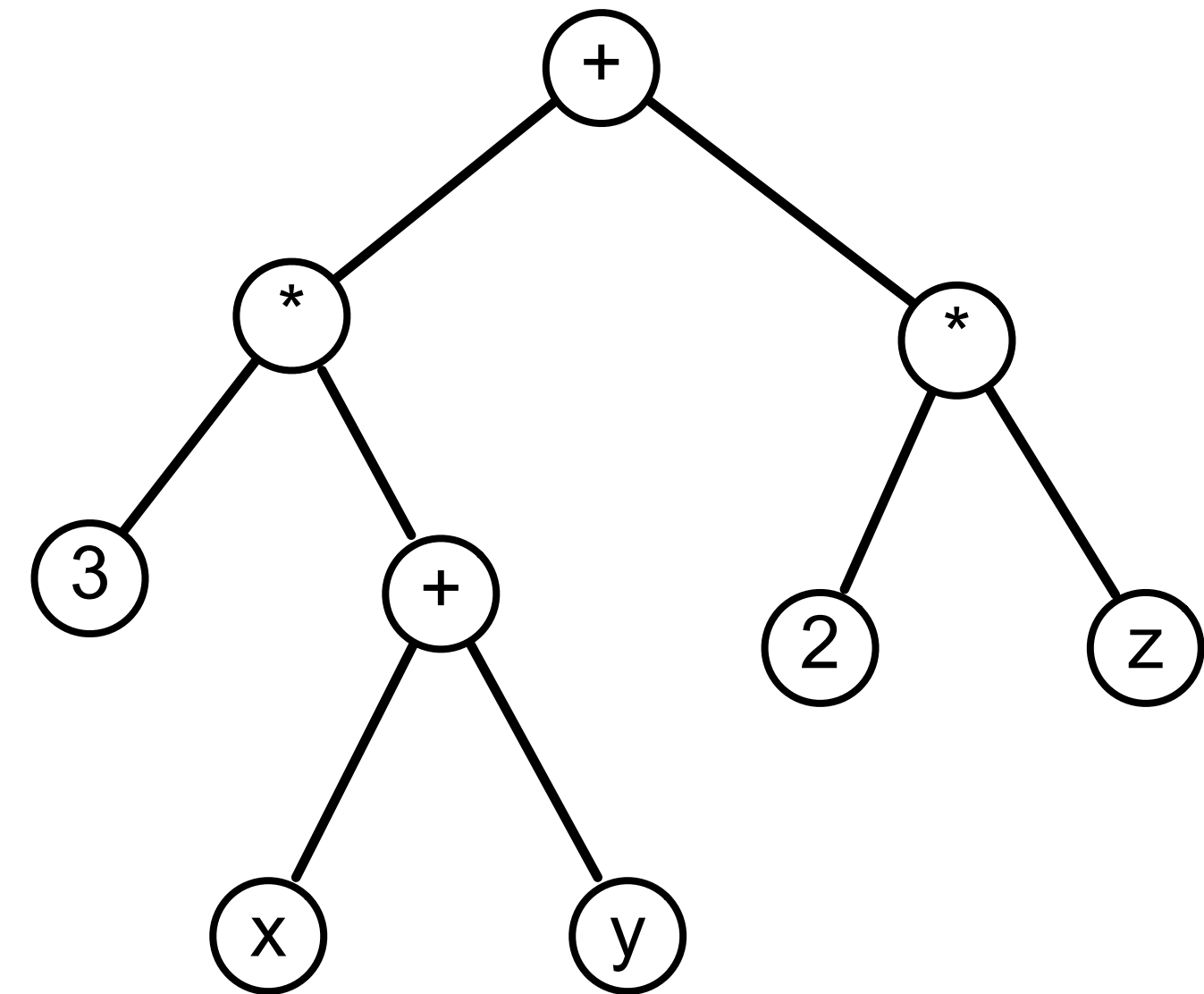
- arbre de syntaxe abstraite représente la structure d'une expression arithmétique
- opérateurs binaires et éléments terminaux (variables, entiers)

```
class BIN_OP:
    def __init__(self, x, g, d) :
        self.val = x
        self.gauche = g
        self.droit = d

    def __str__(self) :
        return "BIN_OP ({{}}, {{}}, {{}})".format(self.val, self.gauche, self.droit)

class ELT:
    def __init__(self, x) :
        self.val = x

    def __str__(self) :
        return "ELT ({{}})".format(self.val)
```



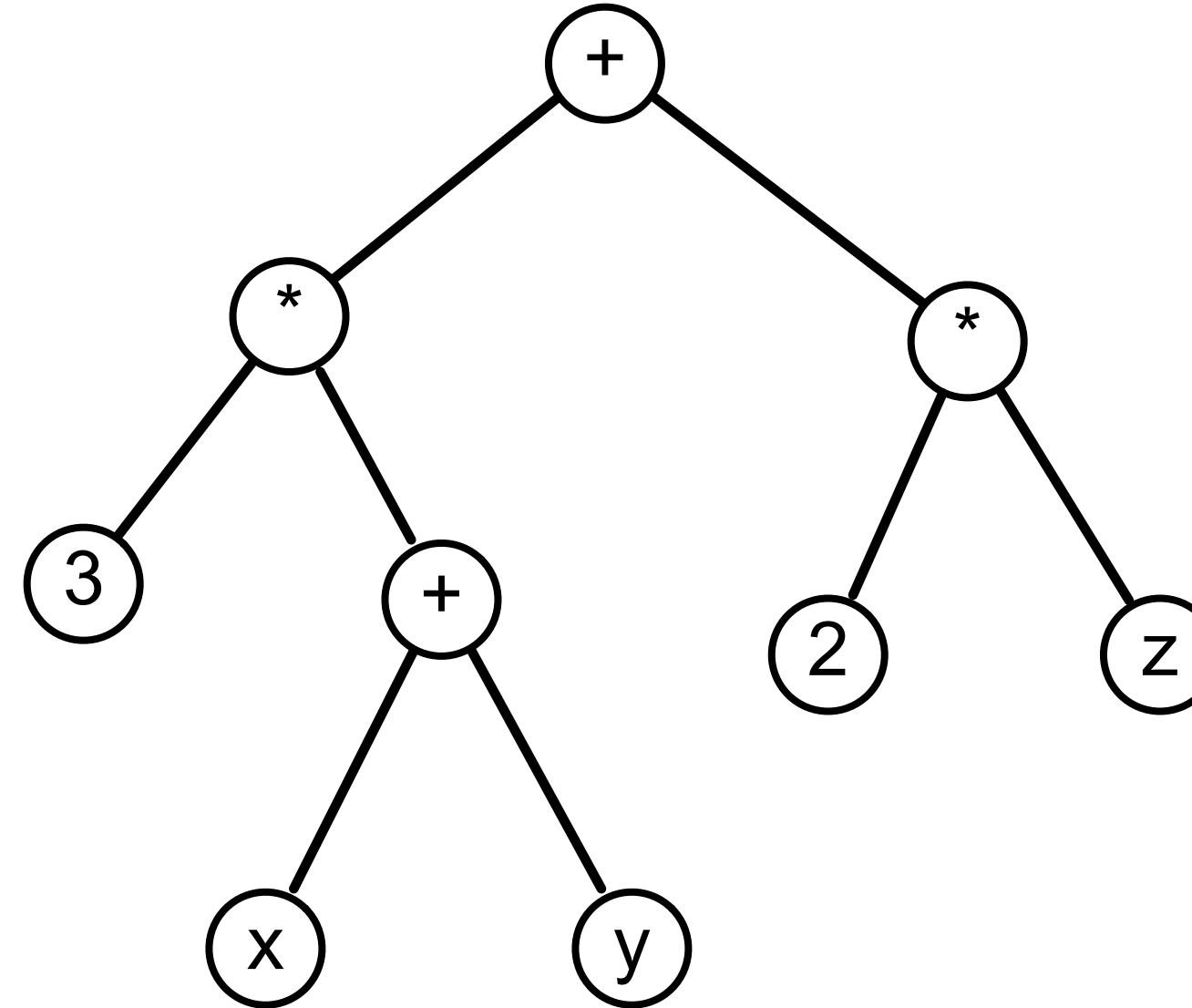
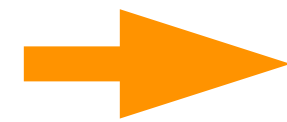
Analyse syntaxique

- compiler le texte pour construire l'ASA correspondant
- analyse lexicale, puis analyse syntaxique (récursive descendante)

```
def compile (txt) :  
    global s, lc  
    s = Stream (txt)  
    lc = next_lexem (s)  
    return (expression())
```

← s, lc stream et lexème courants
(variables globales pour simplifier la programmation)

```
txt = '3 * (x + y) + 2 * z'  
t = compile (txt)  
print (txt, '-->', t)
```



Analyse syntaxique

- compiler le texte pour construire l'ASA correspondant
- analyse lexicale, puis analyse syntaxique (récursive descendante)

```
def avancer () :  
    global s, lc  
    lc = next_lexem(s)
```

```
def expression () :  
    t = produit()  
    (kind, val) = lc  
    if kind == 'OP' and val in {'+', '-'}:  
        avancer()  
        return BIN_OP (val, t, expression())  
    else :  
        return t
```

```
def produit() :  
    t = facteur()  
    (kind, val) = lc  
    if kind == 'OP' and val in {'*', '/'} :  
        avancer()  
        return BIN_OP (val, t, produit())  
    else :  
        return t
```

```
def facteur() :  
    (kind, val) = lc  
    if kind == 'PAR0' :  
        avancer()  
        t = expression()  
        (kind, val) = lc  
        if kind != 'PAR1' :  
            raise Exception ("Il manque ')"")  
    elif kind in {'INT', 'ID'} :  
        t = ELT (val)  
    else :  
        raise Exception ("Erreur de syntaxe")  
    avancer()  
    return t
```

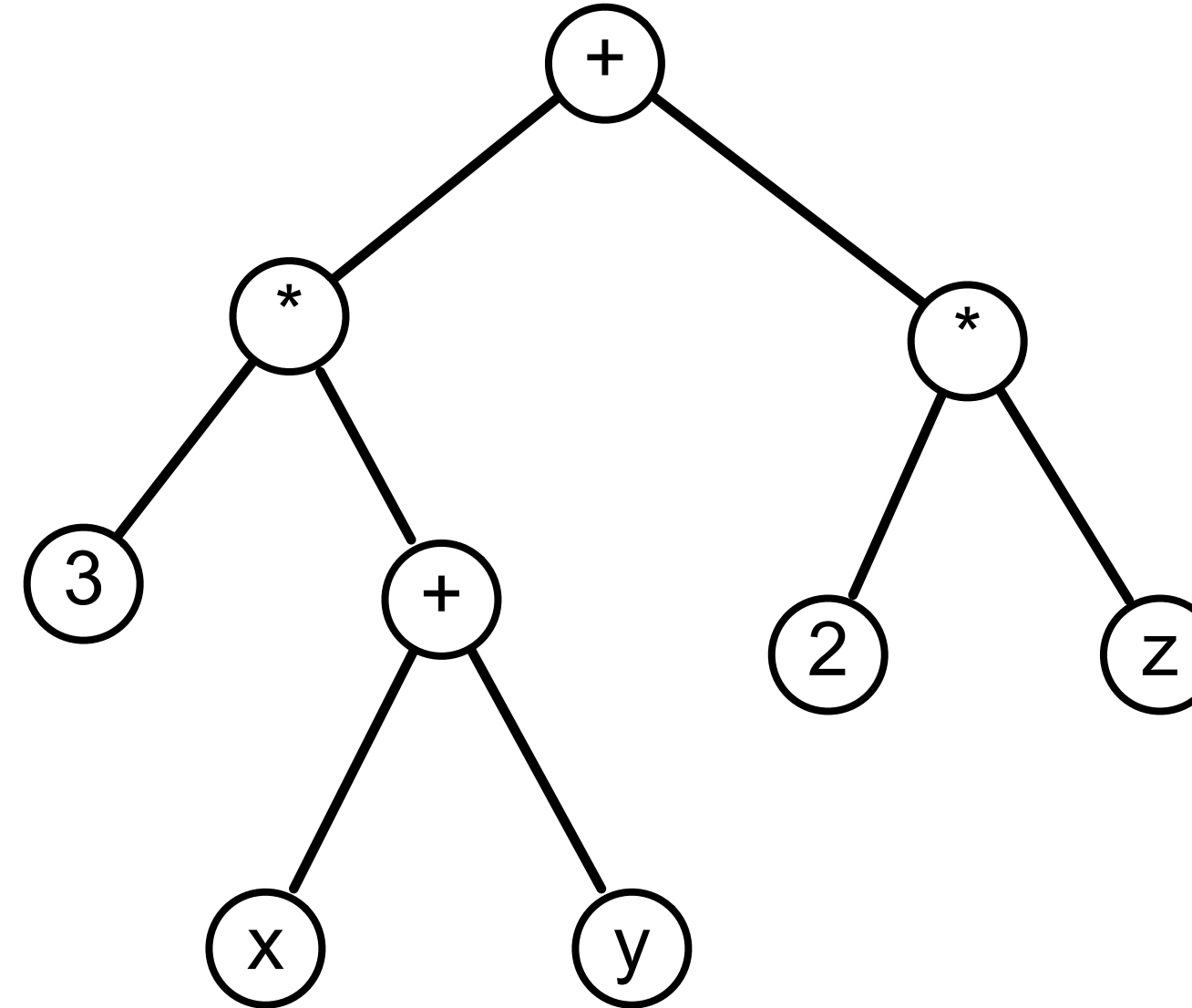
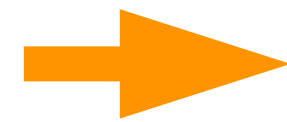

Analyse syntaxique

- compiler le texte pour construire l'ASA correspondant
- analyse lexicale, puis analyse syntaxique (récursive descendante)

```
def compile (txt) :  
    global s, lc  
    s = Stream (txt)  
    lc = next_lexem (s)  
    return (expression())
```

← s, lc stream et lexème courants
(variables globales pour simplifier la programmation)

```
txt = '3 * (x + y) + 2 * z'  
t = compile (txt)  
print (txt, '-->', t)
```

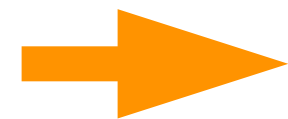


Evaluation

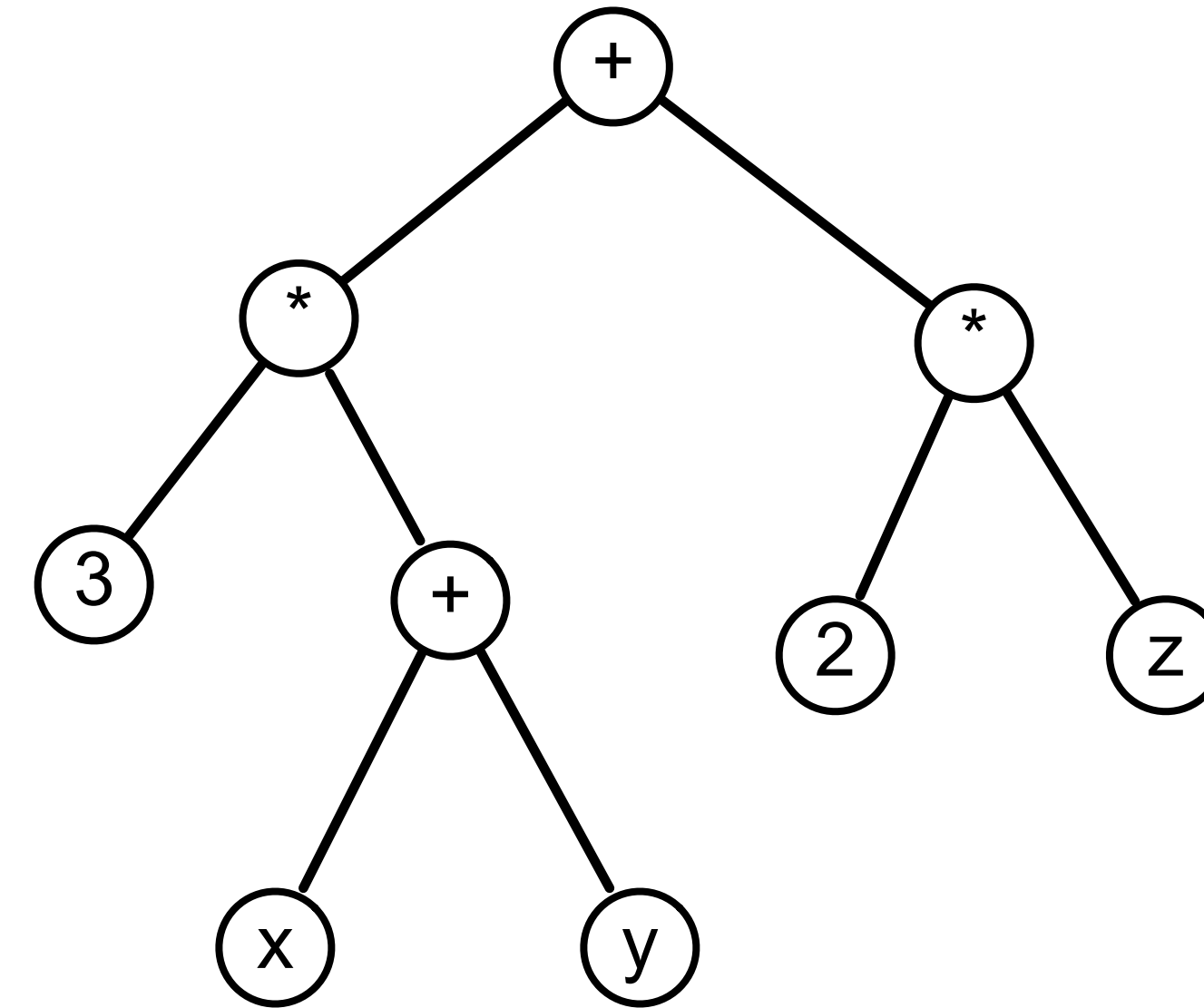
- on calcule l'expression arithmétique dans un environnement où on donne une valeur aux variables

```
def eval (t, e) :  
    if isinstance (t, BIN_OP) :  
        if t.val == '+' :  
            return eval(t.gauche, e) + eval(t.droit, e)  
        elif t.val == '-' :  
            return eval(t.gauche, e) - eval(t.droit, e)  
        elif t.val == '*' :  
            return eval(t.gauche, e) * eval(t.droit, e)  
        elif t.val == '/' :  
            return eval(t.gauche, e) / eval(t.droit, e)  
        else:  
            raise Exception ('BIN_OP impossible')  
    elif isinstance (t.val, int) :  
        return t.val  
    else:  
        return e[t.val]
```

```
env = {'x': 10, 'y':28, 'z':36}  
print (env, '-->', eval (t, env))
```



{'x': 10, 'y': 28, 'z': 36} --> 186



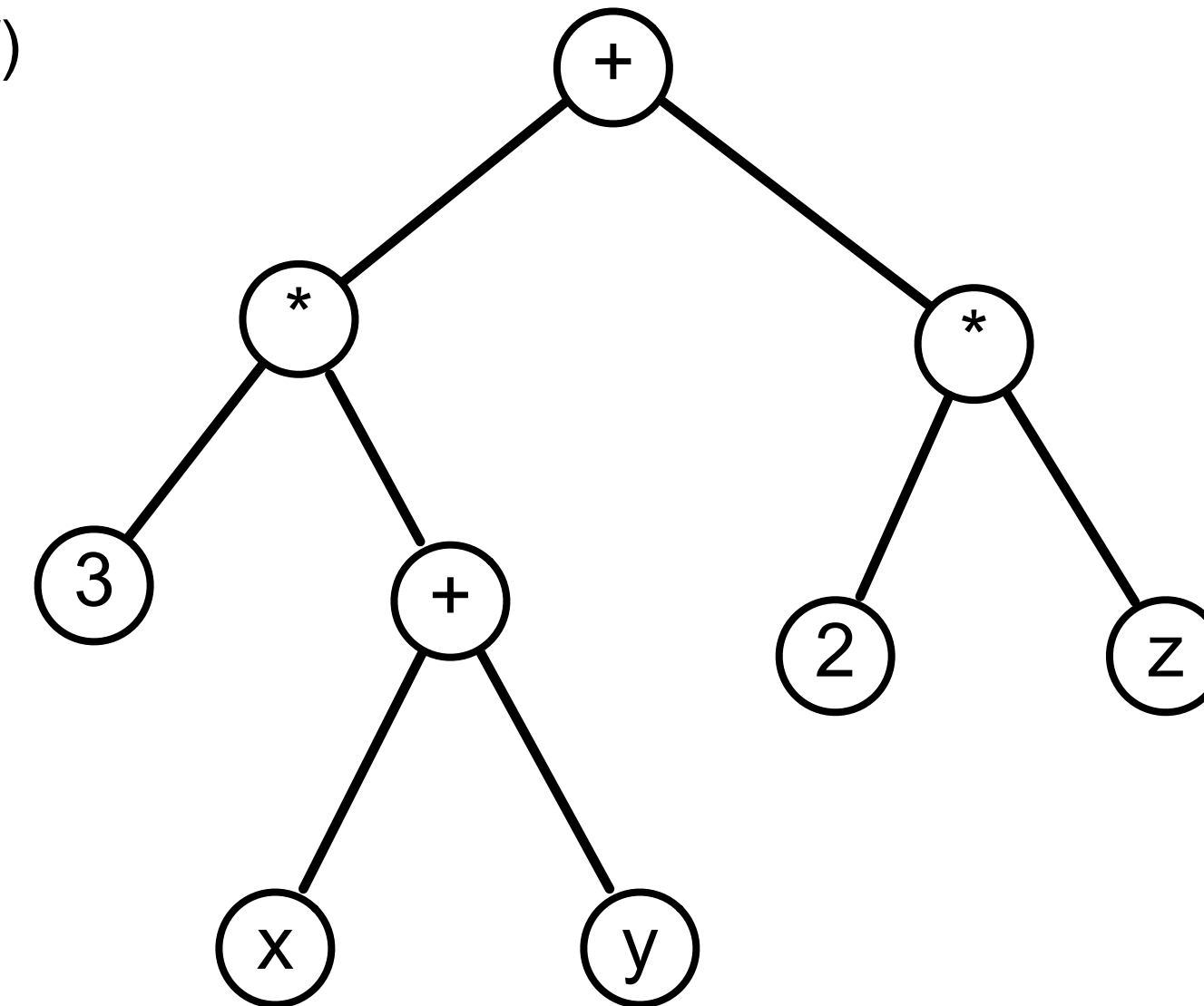
Compilation en langage machine

- une machine a un processeur de calcul (CPU), des registres de calcul et une mémoire

$R_i \leftarrow X$	lire la mémoire X dans un registre (<i>LOAD</i>)
$R_i \leftarrow \text{constante}$	charger une constante dans un registre (<i>LOADI</i>)
$R_i \rightarrow R_i + R_j$	addition (<i>ADD</i>)
$R_i \rightarrow R_i * R_j$	multiplication (<i>MUL</i>)
$R_i \rightarrow X$	rangement dans la mémoire X (<i>STORE</i>)

- l'expression ci-contre peut être compilée dans le programme machine suivant avec 3 registres :

```
R0 ← 3
R1 ← X
R2 ← Y
R1 ← R1 + R2
R0 ← R0 + R1
R1 ← 2
R2 ← R1 * R2
R0 ← R0 + R2
```



- peut-on la compiler avec 2 registres ?

Exercices

Exercice 1 La fonction `expression()` donne les expressions parenthésées implicitement à droite $x + y + 1$ donne $x + (y + 1)$. Comment la changer pour parenthéser implicitement à gauche ?

Exercice 2 Comment faire l'analyse syntaxique d'une série d'initialisations de variables et d'imprimer le résultat sous la forme:

```
let x = 10 in
let y = 3 in
print (x + (3 * y - 5))
```

Exercice 3 Expliquer pourquoi l'analyse syntaxique est plus compliqué pour le faire avec un texte de la forme:

```
x = 10
y = 3
print (x + (3 * y - 5))
```

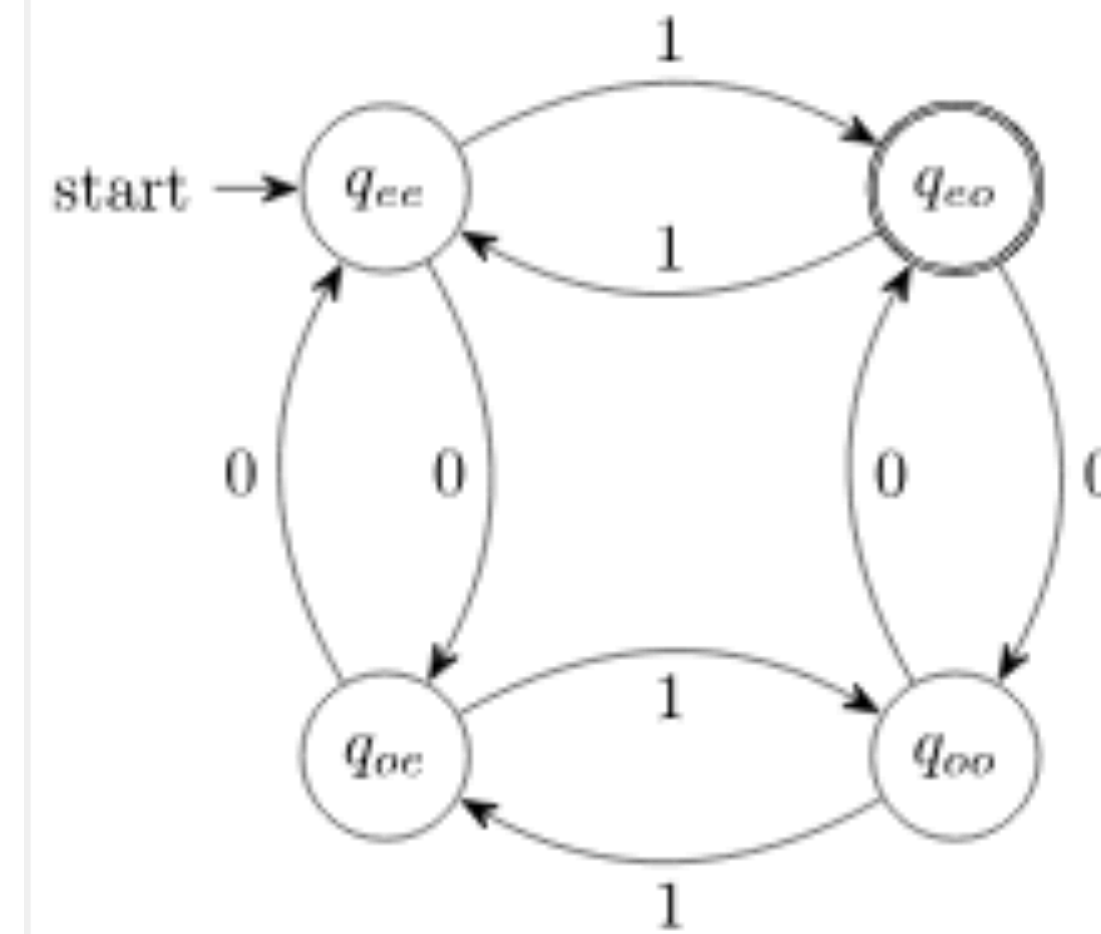
Exercice 4 Programmer la compilation d'expression arithmétique en langage machine ? Comment optimiser le nombre de registres alloués? Que faire avec un nombre de registres insuffisant ?

Théorie

- expressions régulières et **automates finis**

$lettre = a|b|\dots z|A|B|\dots Z|_$
 $chiffre = 0|1|2|3|4|5|6|7|8|9$
 $identificateur = lettre(lettre|chiffre)^*$
 $nombre = chiffre^+$

- des milliers de livres Automata and Computability [D.C. Kozen]



Théorie

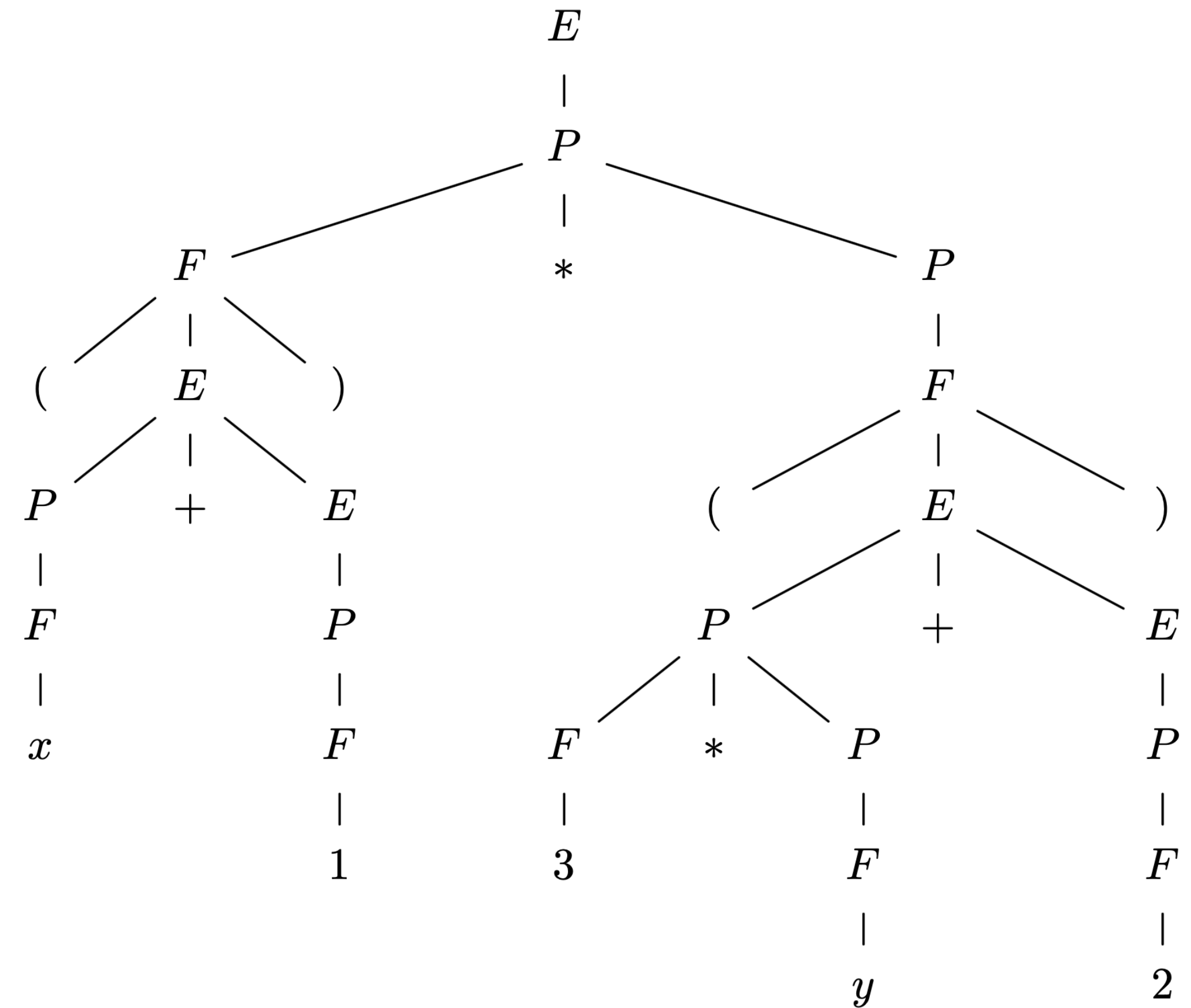
- langages formels (algébriques, rationnels, context-free) [Chomsky - 1956]

$$E \rightarrow P \mid P + E$$

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

$$P \rightarrow F \mid F \times P$$

- arbre syntaxique d'une expression arithmétique
- des milliers d'articles et livres
- école française [Schutzenberger - 1960]



Outils

- le module Python **RegEx** permet de manipuler des expressions régulières
- les modules **PLY** (Python Lex-Yacc)
- **Lex** permet de générer un analyseur lexical à partir d'expressions régulières [M. Lesk - Unix 1975]
- **Yacc** génère un analyseur syntaxique à partir d'une description de grammaire LR(k) [S.C. Johnson - Unix 1975]
 - la méthode récursive descendante a des limitations (pas de récursivité gauche)
 - les méthodes ascendantes LR(k) sont plus générales
- **BNF** (Backus Naur Form) est une forme pour écrire la grammaire formelle des langages de programmation
- la grammaire pour le langage Python se trouve en <https://docs.python.org/3/reference/grammar.html>

Prochain cours

- calcul numérique
- nombres flottants IEEE
- méthode de Newton
- régression linéaire