

# Table des matières

<b>1 Introduction à Python. . . . .</b>	<b>3</b>
1.1 Introduction à PyCharm	3
1.1.1 Installer Python et l'environnement PyCharm	3
1.1.2 Créer un nouveau projet	4
1.1.3 Introduction au débogueur	5
1.2 Saisies, affichages, conversions et calculs	7
1.3 Instruction de choix	9
1.4 Boucles while et for	10
1.4.1 Boucle while	10
1.4.2 Boucle for	11
1.5 Listes	12
1.6 Caractères et chaînes de caractères	15
1.7 Fonctions	17
<b>2 Principes de conception d'un analyseur syntaxique. . . . .</b>	<b>19</b>
2.1 Grammaire d'une langue	19
2.1.1 Définition	19
2.1.2 Analyse syntaxique	20
2.2 Introduction au langage Z	21
2.2.1 Grammaire	21
2.2.2 Passage d'une grammaire à un analyseur syntaxique	26
2.2.3 Analyseur syntaxique	28
<b>3 Principes de conception d'un compilateur . . . . .</b>	<b>32</b>
3.1 Introduction	32
3.2 Première version du langage Z	33
3.2.1 Grammaire	33
3.2.2 Compilateur	33
3.3 Deuxième version du langage Z	35
3.3.1 Grammaire	35
3.3.2 Compilateur	36
3.4 Troisième version du langage Z	39
3.4.1 Grammaire	39
3.4.2 Compilateur	40
<b>4 Principes de conception d'un interpréteur. . . . .</b>	<b>44</b>

---

4.1 Introduction	44
4.2 Première version du langage Z	45
4.2.1 Grammaire	45
4.2.2 Interpréteur	45
4.3 Deuxième version du langage Z	47
4.3.1 Grammaire	47
4.3.2 Interpréteur	47
4.4 Troisième version du langage Z	49
4.4.1 Grammaire	49
4.4.2 Interpréteur	50
<b>Bibliographie. . . . .</b>	<b>54</b>

## Chapitre 1

# Introduction à Python

## 1.1 Introduction à PyCharm

### 1.1.1 Installer Python et l'environnement PyCharm

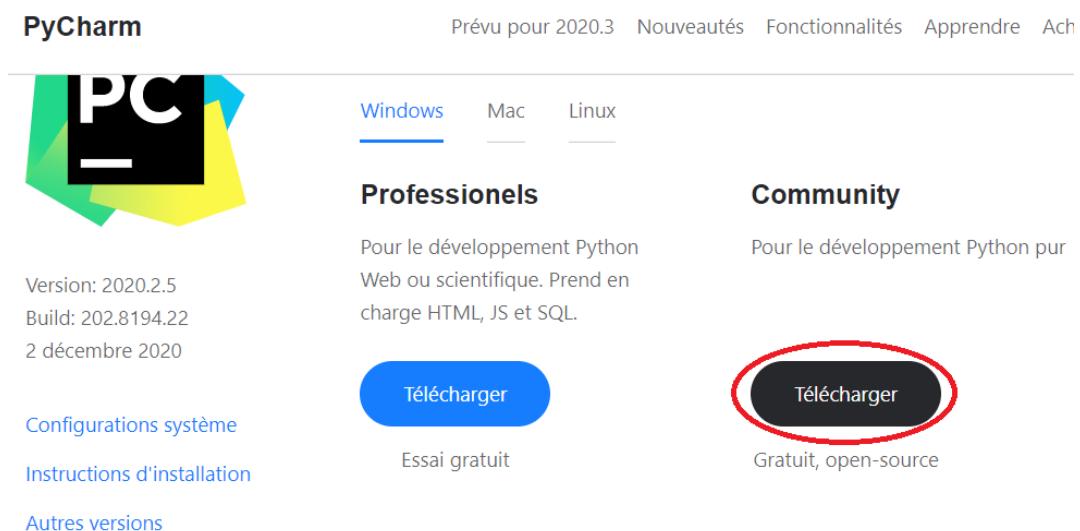
1. Tout d'abord, téléchargez et installez le compilateur **python** :

<https://www.python.org/downloads/>



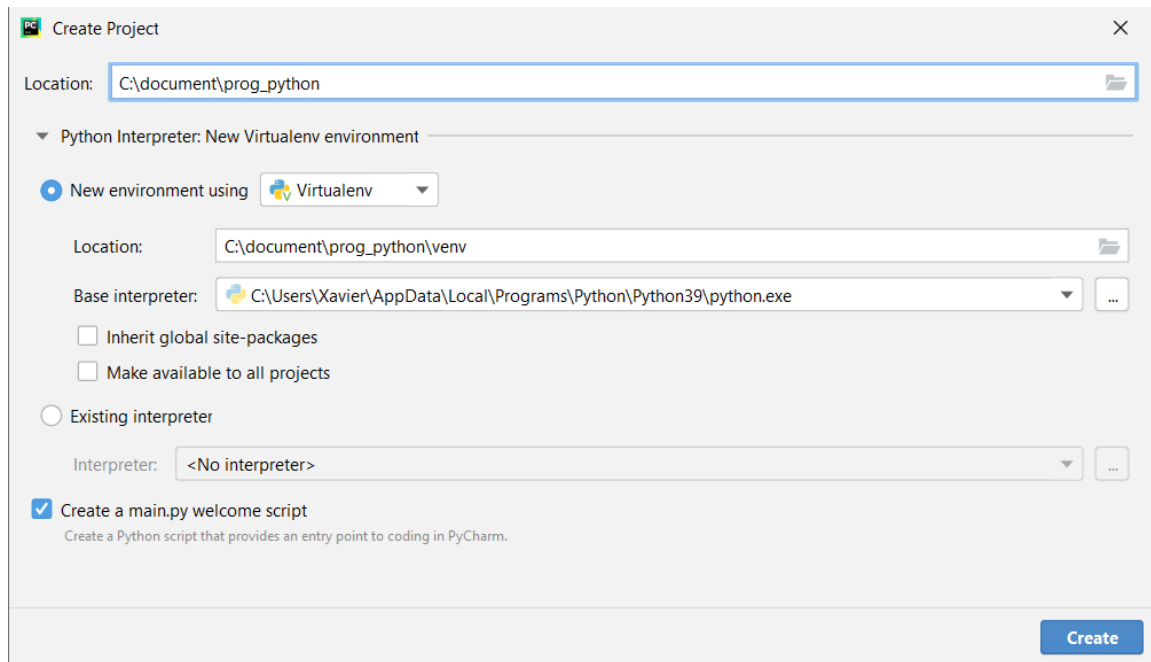
2. Ensuite, téléchargez et installez l'environnement fenêtré **PyCharm Community** :

<https://www.jetbrains.com/fr-fr/pycharm/download/#section=windows>

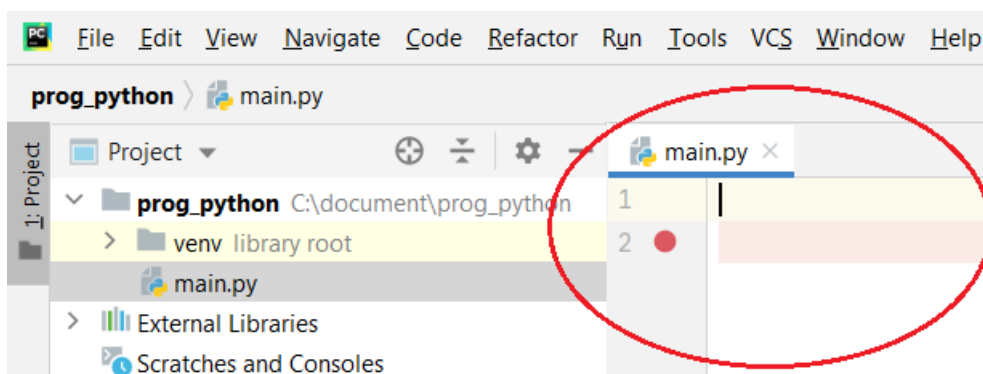


### 1.1.2 Créer un nouveau projet

1. Tout d'abord, créez un projet en sélectionnant *File* → *New Project*.
2. Dans la fenêtre suivante, indiquez où créer le projet et cliquez sur *Create*.

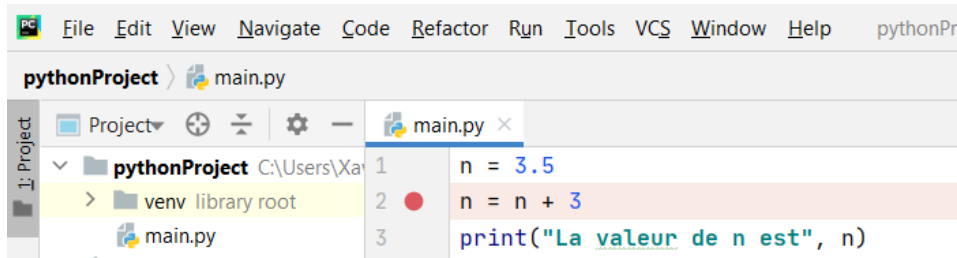


3. Une fois le projet créé, vous pouvez commencer à écrire du code en Python dans la fenêtre centrale à droite. Il s'agit de la fenêtre d'édition du contenu du fichier **main.py** qui est le fichier dans le projet dans lequel se trouve le code source du programme.



### 1.1.3 Introduction au débogueur

1. Pour utiliser le débogueur, placez un point d'arrêt (point rouge) sur la ligne où devra être stoppée temporairement l'exécution du programme.

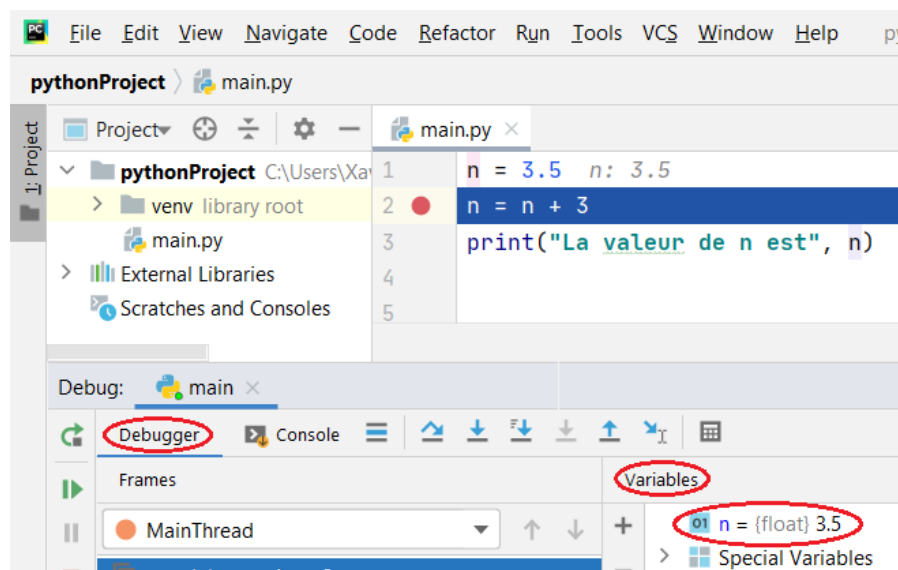


2. Pour démarrer l'exécution du programme dans le débogueur, cliquez sur :

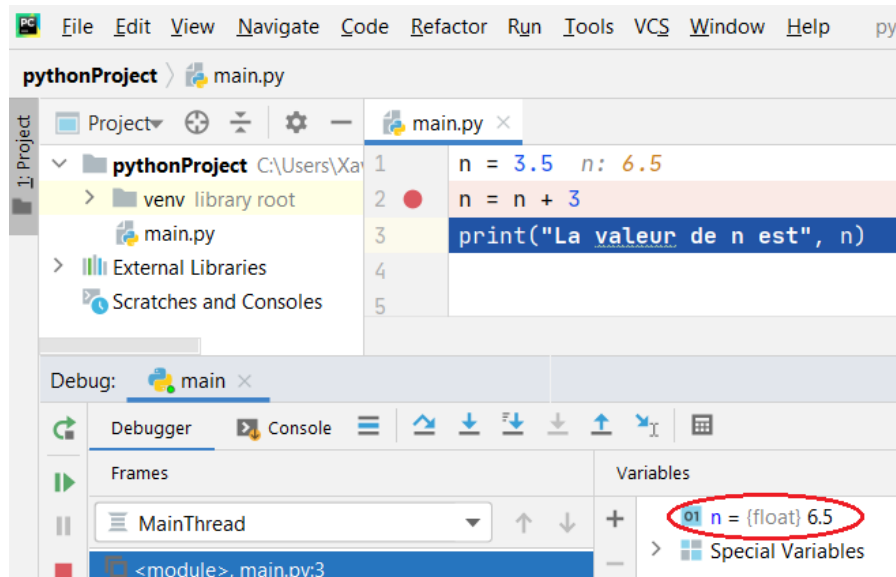


(La flèche verte permet de démarrer une exécution normale du programme.)

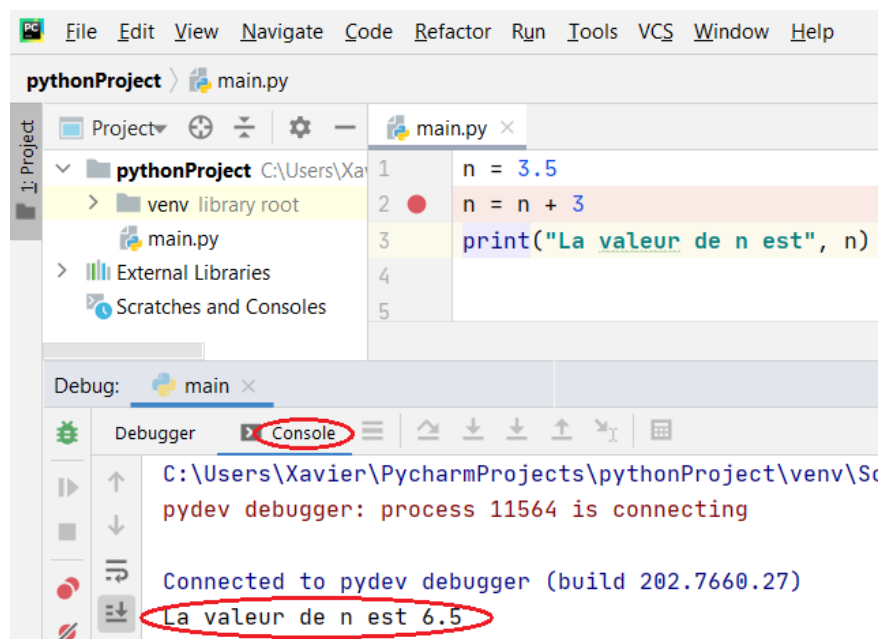
3. L'exécution du programme est stoppée sur la ligne 2. Dans la fenêtre *Debugger* → *Variables*, on aperçoit la valeur de la variable *n* qui est 3.5.



4. Appuyez sur F8 pour exécuter la ligne 2. La valeur de la variable `n`, à présent, est 6.5.



5. Appuyez sur F8 pour exécuter la ligne 3. Le message affiché avec la fonction `print()` se trouve dans la fenêtre *Console*.



**Remarque :** comme alternative à PyCharm, il existe un interpréteur Python en ligne à l'adresse suivante : [https://www.onlinegdb.com/online\\_python\\_compiler](https://www.onlinegdb.com/online_python_compiler)

## 1.2 Saisies, affichages, conversions et calculs

Ex.

```
# tout premier exemple en Python ...

''' ceci est un commentaire ...
    ... sur plusieurs lignes '''

n = float(input("Entrer un flottant : "))
print(n)

n = int(input("Entrer un entier : "))
print(n)

quotfloat = n / 3
quotint = int(n / 3)
resteint = n % 3

chaine = "Les 3 valeurs sont : " + str(quotfloat) + ", " + str(quotint) + ", " + str(resteint)
chaine1 = "Les 3 valeurs sont : {}, {}, {}".format(round(quotfloat, 2), quotint, resteint)

print(chaine)
print(chaine1)
```

Le symbole `#` débute une seule ligne en commentaire. Pour placer en une fois plusieurs lignes en commentaire, il faut utiliser `'''` comme symbole de début et comme symbole de fin de commentaire.

Il n'y a aucune déclaration de variables en Python. Une variable existe aussitôt qu'on assigne une valeur à un identificateur de variable. La variable s'adapte au type de données de la valeur qu'on lui assigne. Dans cet exemple, la variable `n` mémorise d'abord un flottant, puis un entier.

`input()` retourne la chaîne de caractères saisie au clavier.

`print()` affiche des valeurs de types différents. Après chaque usage de `print()`, il y a automatiquement un passage à la ligne suivante.

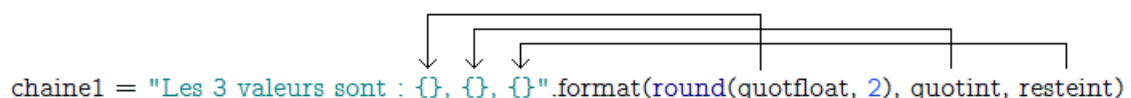
`float()` convertit une valeur (entier, chaîne de caractères, etc.) en un flottant.

`int()` convertit une valeur (flottant, chaîne de caractères, etc.) en un entier.

`str()` convertit une valeur (entier, flottant, etc.) en une chaîne de caractères.

L'opérateur `+` entre 2 chaînes de caractères forme une nouvelle chaîne de caractères qui correspond à la concaténation des 2 chaînes de caractères.

Plutôt que d'utiliser l'opérateur `+` pour former, à partir d'une chaîne de caractères initiale, une nouvelle chaîne de caractères dans laquelle ont été insérées différentes valeurs, on peut utiliser `format()`. Les accolades dans la chaînes de caractères initiale correspondent aux endroits où les valeurs doivent être insérées.



```
chaine1 = "Les 3 valeurs sont : {}, {}, {}".format(round(quotfloat, 2), quotint, resteint)
```

**round()** permet de choisir le nombre de décimales à conserver pour un flottant.

Résultat de l'exécution du code à l'écran :

```
Entrer un flottant : 12.3768
12.3768
Entrer un entier : 14
14
Les 3 valeurs sont : 4.666666666666667, 4, 2
Les 3 valeurs sont : 4.67, 4, 2
```

□

**Ex.**

```
n1 = 3
n2 = 82
n3 = -19


print("les 3 valeurs sont :", n1, ",", n2, ",", n3)
print("les 3 valeurs sont :" + str(n1) + "," + str(n2) + "," + str(n3))
print("les 3 valeurs sont :\n{:4d},\n{:4d},\n{:4d}".format(n1, n2, n3))
```

Si on utilise une virgule entre 2 valeurs dans **print()**, alors un espace est affiché entre ces 2 valeurs. Avec la virgule, chaque valeur numérique peut être affichée sans avoir à être préalablement convertie en une chaîne de caractères.

Si on utilise l'opérateur de concaténation **+**, on crée une nouvelle chaîne de caractères. Les éléments à concaténer doivent être des chaînes de caractères. Quand l'opérateur **+** est utilisé dans **print()**, aucun espace n'est ajouté entre les 2 chaînes de caractères.

Le caractère **\n** dans une chaîne permet, comme en langage C, de passer à la ligne suivante à l'écran.

On peut utiliser **format()** pour définir le nombre de caractères que va occuper un nombre dans une chaîne de caractères après conversion. Ceci permet d'aligner correctement les nombres lors de leur affichage dans des colonnes. Par exemple, l'usage suivant de **format()** crée une nouvelle chaîne de caractères dans laquelle les 3 nombres entiers **n1**, **n2** et **n3**, occuperont chacun 4 caractères.


  
**"les 3 valeurs sont :\n{:4d},\n{:4d},\n{:4d}".format(n1, n2, n3)**

Résultat de l'exécution du code à l'écran :

```
les 3 valeurs sont : 3 , 82 , -19
les 3 valeurs sont :3,82,-19
les 3 valeurs sont :
  3,
 82,
-19
```

□



**Ex.** Le programme suivant affiche 3 flottants entre 0 et 100 (exclu) et 3 entiers entre 0 et 100 (inclu) générés aléatoirement.

```
from random import random
from random import randint

print("Le flottant n°1 généré aléatoirement est", str(round(random() * 100, 2)))
print("La flottant n°2 généré aléatoirement est", str(round(random() * 100, 2)))
print("La flottant n°3 généré aléatoirement est", str(round(random() * 100, 2)))

print("L'entier n°1 généré aléatoirement est", randint(0, 100))
print("L'entier n°2 généré aléatoirement est", randint(0, 100))
print("L'entier n°3 généré aléatoirement est", randint(0, 100))
```

Les fonctions **random()** et **randint()** appartiennent au module **random** et il faut les importer avant de pouvoir les utiliser.

La fonction **random()** retourne un flottant généré aléatoirement tiré de l'intervalle  $0.0 \leq nb < 1.0$ .

La fonction **randint(inf, sup)** retourne un entier généré aléatoirement tiré de l'intervalle  $inf \leq nb \leq sup$ .

Résultat de l'exécution du code à l'écran :

```
Le flottant n°1 généré aléatoirement est 96.61
La flottant n°2 généré aléatoirement est 41.99
La flottant n°3 généré aléatoirement est 71.11
L'entier n°1 généré aléatoirement est 24
L'entier n°2 généré aléatoirement est 41
L'entier n°3 généré aléatoirement est 1
```

□

## 1.3 Instruction de choix

La forme générale de l'instruction **if** est la suivante (les crochets indiquent ce qui n'est pas obligatoire) :

```
if expression booléenne:
    code interne au bloc if
[elif expression booléenne:
    code interne au bloc elif
[elif expression booléenne:
    code interne au bloc elif
...]]
[else:
    code interne au bloc else
]
```

L'instruction **if** peut être constituée d'une seule expression booléenne à évaluer (**if ...**), ou être constituée de plusieurs expressions booléennes à évaluer (**if... elif... elif...**). Elle peut aussi comprendre une partie **else**.

Les opérateurs de comparaison sont les mêmes en Python qu'en langage C : **==**, **!=**, **<=**, **>=**, **<**, et **>**.

Les indentations permettent de créer un bloc d'instructions. En Python, il n'y a ainsi aucun symbole de début et de fin de bloc comme en langage C avec les symboles { }.

**Ex.** Ce programme demande un nombre et il affiche à quel intervalle appartient ce nombre.

```
n = int(input("Entrer un nombre : "))

if n < 0:
    print("Le nombre entré est négatif")
elif n == 0:
    print("le nombre entré est zéro")
else:
    print("Le nombre entré est positif")
```

Résultat de l'exécution du code à l'écran :

```
Entrer un nombre : 45
Le nombre entré est positif
```

□

**Ex.** Pour connecter ensemble des expressions booléennes simples on peut utiliser les opérateurs conditionnels. L'opérateur conditionnel et s'écrit **and** et l'opérateur conditionnel ou s'écrit **or**.

```
n = int(input("Valeur : "))

if n > 0 and n < 10:
    print("la valeur de n fait partie de l'intervalle [1, 9]")

if n <= 0 or n >= 10:
    print("la valeur de n ne fait pas partie de l'intervalle [1, 9]")
```

L'opérateur conditionnel et permet notamment de tester si une valeur appartient à un intervalle de valeurs, dans cet exemple [0, 9], et l'opérateur conditionnel ou permet notamment de tester si une valeur n'appartient pas à un intervalle de valeurs.

Résultat de l'exécution du code à l'écran :

```
Valeur : 12
la valeur de n ne fait pas partie de l'intervalle [1, 9]
```

□

## 1.4 Boucles while et for

### 1.4.1 Boucle while

La forme générale de la boucle **while** est la suivante :

```
while expression booléenne:
    code interne à la boucle
```

Les indentations identifient le code interne à la boucle.

**Ex.** Ce programme affiche tous les diviseurs d'un nombre.

```
nb = int(input("Entrez un nombre : "))  
  
i = 1  
  
while i <= nb:  
    if nb % i == 0:  
        print(i)  
    i = i + 1
```

Résultat de l'exécution du code à l'écran :

```
Entrez un nombre : 6  
1  
2  
3  
6
```

□

### 1.4.2 Boucle for

La forme générale de la boucle **for** est la suivante (les crochets indiquent ce qui n'est pas obligatoire) :

```
for var in range(nombre de départ, nombre de fin [, pas]) :  
    code interne à la boucle
```

**range()** retourne à chaque tour de boucle un nombre entier qui est assigné à la variable *var*. Ce nombre est déterminé ainsi :

- Si *pas* est positif, la variable *var* évolue à chaque tour de boucle ainsi :  $var = var + pas$ . La boucle se termine dès que **range()** retourne un nombre  $\geq$  nombre de fin. Le dernier nombre n'est pas assigné à *var*. Au cours du temps, le nombre assigné à *var* provient donc de la séquence de nombres : nombre de départ  $\leq$  nombre  $<$  nombre de fin.
- Si *pas* est négatif, la variable *var* évolue à chaque tour de boucle ainsi :  $var = var - pas$ . La boucle se termine dès que **range()** retourne un nombre  $\leq$  nombre de fin. Le dernier nombre n'est pas assigné à *var*. Au cours du temps, le nombre assigné à *var* provient donc de la séquence de nombres : nombre de départ  $\geq$  nombre  $>$  nombre de fin.

Par défaut, le *pas* est +1.

Les indentations identifient le code interne à la boucle.

**Ex.** Ce programme affiche les valeurs impaires comprise entre 1 et 10.

```
for i in range(1, 10, 2):  
    print("Valeur de i =", i)
```

La variable *i* stocke initialement la valeur 1. La valeur de cette variable est augmentée de 2 à

chaque tour de boucle jusqu'à atteindre la valeur 9.

Résultat de l'exécution du code à l'écran :

```
Valeur de i = 1
Valeur de i = 3
Valeur de i = 5
Valeur de i = 7
Valeur de i = 9
```

Cette boucle for correspond à la boucle while suivante :

```
i = 1;

while i < 10:
    print("Valeur de i =", var)
    i = i + 2

i = i - 2
```

□

**Ex.** Ce programme affiche les valeurs entre 0 et 9 par ordre croissant, puis par ordre décroissant.

```
for i in range(0, 10):
    print(str(i) + " ", end='')

print("")

for i in range(9, -1, -1):
    print(str(i) + " ", end='')
```

Dans `print()`, le paramètre `end = "` permet de n'avoir pas de passage à la ligne suivante à la fin de l'affichage.

Résultat de l'exécution du code à l'écran :

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

□

## 1.5 Listes

**Ex.**

```
Mots = ["Vélo", "Voiture", "Arbre", "Fenêtre"]

print("Mots =", Mots)
print("Nombre d'éléments =", len(Mots))

Mots.append("Fleur")

print("Mots =", Mots)
print("Nombre d'éléments =", len(Mots))

Mots.remove("Voiture")
```

```
print("Mots =", Mots)
print("Nombre d'éléments =", len(Mots))

print(Mots[0])
```

La liste dans cet exemple est une liste de chaînes de caractères.

`len(Mots)` retourne le nombre d'éléments se trouvant dans la liste `Mots`.

`Mots.append("Fleur")` ajoute le mot "Fleur" à la fin de la liste `Mots`.

`Mots.remove("Voiture")` supprime le mot "Voiture" de la liste `Mots`.

Dans une liste, la position 0 est celle du 1<sup>er</sup> élément.

Résultat de l'exécution du code à l'écran :

```
Mots = ['Vélo', 'Voiture', 'Arbre', 'Fenêtre']
Nombre d'éléments = 4
Mots = ['Vélo', 'Voiture', 'Arbre', 'Fenêtre', 'Fleur']
Nombre d'éléments = 5
Mots = ['Vélo', 'Arbre', 'Fenêtre', 'Fleur']
Nombre d'éléments = 4
Vélo
```

□

**Ex.**

```
Valeurs = [ ]

print("Nombre d'éléments =", len(Valeurs))

Valeurs.append(29.3)

print("Valeurs =", Valeurs)
print("Nombre d'éléments =", len(Valeurs))

Valeurs.append(12.87)

print("Valeurs =", Valeurs)
print("Nombre d'éléments =", len(Valeurs))

Valeurs.insert(1, 11.2)

print("Valeurs =", Valeurs)
print("Nombre d'éléments =", len(Valeurs))

dernier = Valeurs.pop()

print("Dernier élément =", dernier)
print("Valeurs =", Valeurs)
print("Nombre d'éléments =", len(Valeurs))

premier = Valeurs.pop(0)

print("Premier élément =", premier)
print("Valeurs =", Valeurs)
print("Nombre d'éléments =", len(Valeurs))
```

**Valeurs** = [] crée une liste qui ne contient aucun élément.

**Valeurs.append(29.3)** ajoute le nombre 29.3 à la liste Valeurs.

**Valeurs.insert(1, 11.2)** ajoute la valeur 11.2 à la position 1 dans la liste Valeurs et la valeur 12.87 est décalée à la position 2 dans cette liste.

**Valeurs.pop()** retire le dernier élément de la liste Valeurs. Elle a le comportement opposé à **Valeurs.append()**. On peut transmettre en paramètre à la fonction **pop()** la position de l'élément à retirer de la liste.

Résultat de l'exécution du code à l'écran :

```
Nombre d'éléments = 0
Valeurs = [29.3]
Nombre d'éléments = 1
Valeurs = [29.3, 12.87]
Nombre d'éléments = 2
Valeurs = [29.3, 11.2, 12.87]
Nombre d'éléments = 3
Dernier élément = 12.87
Valeurs = [29.3, 11.2]
Nombre d'éléments = 2
Premier élément = 29.3
Valeurs = [11.2]
Nombre d'éléments = 1
```

□

**Ex.**

```
Valeurs = [2, 4, 9, -3, 98, -24, -48]

for i in range(len(Valeurs)):
    Valeurs[i] = Valeurs[i] + 1

print("Valeurs =", Valeurs)

Valeurs1 = Valeurs[1 : 4]

print("Valeurs1 =", Valeurs1)

n = int(input("Entrer un nombre : "))

if n in Valeurs1: print("La valeur", n, "est dans la liste Valeurs1")
else: print("La valeur", n, "n'est pas dans la liste Valeurs1")
```

On peut voir une liste en Python comme un vecteur en langage C. La boucle au début du programme incrémente la valeur de chaque nombre de la liste Valeurs.

Avec **Valeurs1 = Valeurs[1 : 4]**, on crée la liste Valeurs1 qui a, pour contenu, une copie des nombres de la liste Valeurs situés aux positions 1, 2 et 3.

Avec **if n in Valeurs1**, on teste si le nombre n est présent dans la liste Valeurs1.

Résultat de l'exécution du code à l'écran :

```
Valeurs = [3, 5, 10, -2, 99, -23, -47]
Valeurs1 = [5, 10, -2]
Entrer un nombre : -2
La valeur -2 est dans la liste Valeurs1
```

□

**Ex.**

```
liste = []

liste.append(["prénom", "Marcel"])
liste.append(["age", 20])
liste.append(["taille (cm)", 180])

for i in range(len(liste)):
    print("catégorie :", liste[i][0], "; valeur :", liste[i][1])

print("")

for element in liste:
    print("catégorie :", element[0], "; valeur :", element[1])
```

On peut avoir une liste dans laquelle chaque élément est constitué de plusieurs valeurs. Dans cet exemple, chaque élément de la liste est constitué de 2 valeurs : une chaîne de caractères et une valeur numérique. Le contenu de la liste est affiché en utilisant 2 types de boucle différents.

Résultat de l'exécution du code à l'écran :

```
catégorie : prénom ; valeur : Marcel
catégorie : age ; valeur : 20
catégorie : taille (cm) ; valeur : 180

catégorie : prénom ; valeur : Marcel
catégorie : age ; valeur : 20
catégorie : taille (cm) ; valeur : 180
```

□

## 1.6 Caractères et chaînes de caractères

**Ex.** Dans ce programme, si la lettre entrée est majuscule, elle est convertie en minuscule et si c'est une lettre minuscule, elle est convertie en majuscule.

```
c = input("Entrer une lettre : ")

if c.isupper():
    c = c.lower()
else:
    c = c.upper()

print("La lettre est maintenant : " + c)
```

Résultat de l'exécution du code à l'écran :

```
Entrer une lettre : a
La lettre est maintenant : A
```

□

**Ex.**

```
prenom = input("Entrer votre prénom : ")
debut = input("Entrer les 2 lettres à tester avec les 2 premières lettres du prénom : ")

if prenom[0 : 2] == debut:
    print("Le prénom commence bien avec", debut)
else:
    print("Le prénom ne commence pas avec", debut)
```

Pour extraire une sous-chaîne d'une chaîne de caractères, on indique, entre crochets après le nom de la chaîne, la position du caractère de début et la position du caractère situé immédiatement après le caractère final. Par exemple, avec **prenom[0 : 2]**, on extrait de la chaîne de caractères prenom la sous-chaîne formée par les 2 premiers caractères (les caractères aux positions 0 et 1).

Pour comparer 2 chaînes de caractères, on utilise l'opérateur **==** dans l'instruction **if**.

Résultat de l'exécution du code à l'écran :

```
Entrer votre prénom : Marcel
Entrer les 2 lettres à tester avec les 2 premières lettres du prénom : Ma
Le prénom commence bien avec Ma
```

□

**On ne peut modifier une chaîne de caractères directement comme on le ferait en langage C.** Pour modifier une chaîne de caractères, il faut créer, à partir de la chaîne de caractères d'origine, une nouvelle chaîne de caractères dans laquelle figure la modification souhaitée.

**Ex.** Voici 2 versions différentes d'un programme qui remplace toutes les occurrences du caractère a par le caractère \* dans la chaîne de caractères ch.

**1<sup>re</sup> version :**

```
ch = "Il fait beau"

print(ch)

for i in range(len(ch)):
    if ch[i] == 'a':
        ch = ch[0 : i] + '*' + ch[i + 1: len(ch)]

print(ch)
```

Avec **ch[0 : i] + '\*' + ch[i + 1: len(ch)]**, on forme une chaîne comprenant les caractères qui figurent avant le caractère a, le caractère \* et les caractères qui suivent le caractère a.



Résultat de l'exécution du code à l'écran :

```
Il fait beau
Il f*it be*u
```

2<sup>e</sup> version :

```
ch = "Il fait beau"
ch1 = ''

print(ch)

for c in ch:
    if c == 'a':
        ch1 = ch1 + '*'
    else:
        ch1 = ch1 + c

ch = ch1

print(ch)
```

Avec la boucle **for c in ch**, on récupère un par un dans la variable **c** les caractères de la chaîne de caractères **ch**.

□

## 1.7 Fonctions

La forme générale d'une fonction est la suivante (les crochets indiquent ce qui n'est pas obligatoire):

```
# définition de La fonction

def MaFonction([param1 [, param2 [, ...]]]):
    code de la fonction
    [return valeur]

# appel à La fonction

MaFonction(...)
```

Une fonction peut recevoir de 0 à plusieurs paramètres.

Les indentations identifient le code interne à la fonction.

**return** permet à une fonction de retourner une valeur.

**Ex.** Le programme suivant contient une fonction qui utilise une variable globale et une valeur passée en paramètre :

```
def AjouterValeur(ajout):
    global var          # var existe déjà et est une variable globale

    var = var + ajout
```

```
var = 3          # var est une variable globale, car elle est créée en dehors d'une fonction
AjouterValeur(5)
print(var)
```

Pour permettre à une fonction d'avoir connaissance de l'existence d'une variable globale, on utilise **global** devant le nom de cette variable au début du code de la fonction. □

**Ex.** Voici 2 versions différentes d'un programme qui compte le nombre de fois qu'une lettre est présente dans une chaîne de caractères.

**1<sup>re</sup> version :**

```
def NbFoisCarDansChaine(chaine, lettre):
    nbfois = 0

    for i in range(len(chaine)):
        if chaine[i] == lettre:
            nbfois = nbfois + 1

    return nbfois

ch = input("Chaîne : ")
c = input("Lettre : ")

print("La chaîne \"" + ch + "\" contient", NbFoisCarDansChaine(ch, c), "fois '" + c + "'")
```

Résultat de l'exécution du code à l'écran :

```
Chaîne : il y a beaucoup de soleil aujourd'hui
Lettre : a
La chaîne "il y a beaucoup de soleil aujourd'hui" contient 3 fois 'a'
```

**2<sup>e</sup> version :**

```
def NbFoisCarDansChaine(chaine, lettre):
    nbfois = 0

    for c in chaine:
        if c == lettre:
            nbfois = nbfois + 1

    return nbfois

ch = input("Chaîne : ")
c = input("Lettre : ")

print("La chaîne \"" + ch + "\" contient", NbFoisCarDansChaine(ch, c), "fois '" + c + "'")
```

□

## Chapitre 2

# Principes de conception d'un analyseur syntaxique

## 2.1 Grammaire d'une langue

### 2.1.1 Définition

La grammaire d'une langue sert à déterminer si une phrase est correctement formée dans cette langue ou non.

Une grammaire est constituée d'une série de règles et chacune a la forme  $\langle X \rangle \rightarrow \alpha$ . Ceci se lit " $\alpha$  est un  $\langle X \rangle$ ".

- $\langle X \rangle$  est la **tête de la règle**. C'est une catégorie abstraite appelée symbole non-terminal.
- $\alpha$  est le **corps de la règle**. Il est constitué de symboles non-terminaux et/ou symboles terminaux. Les symboles terminaux constituent le vocabulaire de la langue.
- La flèche sépare la tête du corps de la règle.

Tout symbole non-terminal est placé entre chevrons pour le distinguer d'un symbole terminal.

**Ex.** Prenons une grammaire simple constituée des quelques règles suivantes :

$\langle \text{phrase} \rangle$	$\rightarrow$	$\langle \text{sujet} \rangle$	$\langle \text{verbe} \rangle$
$\langle \text{sujet} \rangle$	$\rightarrow$	Georges	
$\langle \text{sujet} \rangle$	$\rightarrow$	Victoria	
$\langle \text{verbe} \rangle$	$\rightarrow$	mange	
$\langle \text{verbe} \rangle$	$\rightarrow$	dort	

Dans cette petite grammaire, on trouve :

- Les symboles non-terminaux  $\langle \text{phrase} \rangle$ ,  $\langle \text{sujet} \rangle$  et  $\langle \text{verbe} \rangle$ .
- Les symboles terminaux Georges, Victoria, mange et dort.

Les corps de règle associés à une même tête peuvent être écrits sur la même ligne. Le symbole | qui se lit "ou" sépare alors un corps de règle d'un autre.

La grammaire peut alors s'écrire ainsi :

```

<phrase> → <sujet> <verbe>
<sujet>  → Georges | Victoria
<verbe> → mange | dort

```

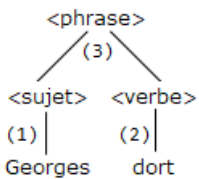
□

### 2.1.2 Analyse syntaxique

À partir de la grammaire d'une langue, pour vérifier qu'une phrase est correctement formée dans cette langue ou non, on procède ainsi :

1. On prend, en tant que point de départ, les symboles terminaux qui constituent la phrase.
2. En appliquant les règles adéquates de la grammaire, progressivement, on réduit (remplace) les symboles terminaux qui constituent la phrase en des symboles non-terminaux et les symboles non-terminaux sont eux-mêmes réduits (remplacés) par des symboles non-terminaux de plus en plus abstraits.
3. Si on réussit à réduire l'entièreté de la phrase en le symbole non-terminal le plus abstrait dans la grammaire, alors la vérification est un succès. Dans le cas contraire, la vérification est un échec et ceci indique que la phrase ne respecte pas la grammaire de la langue.

**Ex.** En utilisant la petite grammaire figurant dans la page précédente, nous allons vérifier si la phrase *Georges dort* est correctement formée.

Application des règles		Arbre syntaxique
Georges dort	(symboles terminaux)	 <pre> graph TD     A["&lt;phrase&gt; (3)"] --&gt; B["&lt;sujet&gt;"]     A --&gt; C["&lt;verbe&gt;"]     B -- "(1)" --&gt; D["Georges"]     C -- "(2)" --&gt; E["dort"] </pre>
<sujet> dort	(<sujet> → Georges) (1)	
<sujet> <verbe>	(<verbe> → dort) (2)	
<phrase>	(<phrase> → <sujet> <verbe>) (3)	

Dans la colonne de gauche du tableau, une règle spécifique de la grammaire est appliquée à chaque ligne. Pour appliquer une règle de la grammaire, il suffit de remplacer le corps de cette règle par sa tête. Par exemple, le symbole terminal Georges est réduit en le symbole non-terminal <sujet> lorsque est appliquée la règle <sujet> → Georges (cette règle se lit "Georges est un <sujet>"). À partir des symboles terminaux Georges dort, 3 règles de la grammaire ont été successivement appliquées dans le but d'atteindre le symbole non-terminal le plus abstrait qui est <phrase>. Georges dort est une phrase correctement formée.

Dans la colonne de droite du tableau, figure un arbre appelé **arbre syntaxique**. Il donne une représentation de l'application des règles. □

## 2.2 Introduction au langage Z

Nous allons appeler langage "Z" le langage de programmation décrit dans ces notes de cours.

### 2.2.1 Grammaire

La grammaire d'un langage de programmation sert à déterminer si un programme exprimé dans ce langage est correctement formé ou non. Chaque langage de programmation : C, Java, PHP, etc., a une grammaire qui lui est propre.

Voici la grammaire du langage Z permettant de reconnaître des expressions arithmétiques constituées de nombres entiers ainsi que des opérateurs \* (multiplication) et + (addition). Cette grammaire doit permettre l'acceptation d'expressions comme :  $5 + 12 * 4 + 256$ ,  $45 * 12 + 16$ ,  $132 * 767$ , ..., et elle doit permettre le rejet d'expressions comme :  $5 \& 56$ ,  $0b466 + xyz / , 8 + * \}$  6676, ...

<code>&lt;prog&gt;</code>	$\rightarrow$	<code>&lt;exprplus&gt;</code>
<code>&lt;exprplus&gt;</code>	$\rightarrow$	<code>&lt;exprfois&gt;</code>   <code>&lt;exprplus&gt; + &lt;exprfois&gt;</code>
<code>&lt;exprfois&gt;</code>	$\rightarrow$	<code>&lt;nombre&gt;</code>   <code>&lt;exprfois&gt; * &lt;nombre&gt;</code>
<code>&lt;nombre&gt;</code>	$\rightarrow$	<code>&lt;chiffre&gt;</code>   <code>&lt;nombre&gt; &lt;chiffre&gt;</code>
<code>&lt;chiffre&gt;</code>	$\rightarrow$	<code>0</code>   <code>1</code>   <code>2</code>   <code>3</code>   <code>4</code>   <code>5</code>   <code>6</code>   <code>7</code>   <code>8</code>   <code>9</code>

Description :

- Cette grammaire permet de reconnaître une expression arithmétique pouvant contenir des nombres en décimal et les opérateurs binaires + et \*.
- Cette grammaire prend en compte la **priorité des opérateurs**. Pour faire savoir que l'opérateur + est moins prioritaire que l'opérateur \*, on associe l'opérateur + à un symbole non-terminal plus abstrait que le symbole non-terminal auquel est associé l'opérateur \*. Ainsi, en lisant les règles de la grammaire, on voit que le symbole non-terminal `<exprplus>` est plus abstrait que le symbole non-terminal `<exprfois>`, car il est plus proche du symbole non-terminal le plus abstrait `<prog>`.
- On a de la **rékursivité** dans une règle de grammaire lorsque le même symbole non-terminal est présent en tête et dans le corps de cette règle. La rékursivité rend possible la répétition un nombre quelconque de fois d'un motif similaire. Plus concrètement, voici les règles de la grammaire dans lesquelles on trouve de la rékursivité :
  - La règle `<nombre>  $\rightarrow$  <nombre> <chiffre>` signifie que tout nombre suivi d'un chiffre est un nombre. Cette règle permet d'avoir un nombre constitué d'un nombre quelconque de chiffres.

**Ex.** Le nombre 521 est constitué de 3 chiffres, le nombre 45491 est constitué de 5 chiffres, etc. □

  - La règle `<exprfois>  $\rightarrow$  <exprfois> * <nombre>` permet d'avoir une expression dans laquelle l'opérateur de multiplication est utilisé un nombre quelconque de fois.

**Ex.** Dans l'expression  $5 * 7845 * 8 + 2$ , on a l'expression  $5 * 7845 * 8$  dans laquelle l'opérateur de multiplication est utilisé à 2 reprises. Dans l'expression  $45 * 12 + 55645 * 28 * 773 * 17 + 45 + 8$ , on a l'expression  $45 * 12$  dans laquelle l'opérateur de multiplication est utilisé à 1 reprise, ainsi que l'expression  $55645 * 28 * 723 * 17$  dans laquelle l'opérateur de multiplication est utilisé à 3 reprises.

□

- La règle  $\langle \text{exprplus} \rangle \rightarrow \langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle$  permet d'avoir une expression dans laquelle l'opérateur d'addition est utilisé un nombre quelconque de fois.

**Ex.** Dans l'expression  $5 * 7845 * 8 + 2$ , l'opérateur d'addition est utilisé à 1 reprise. Dans l'expression  $45 * 12 + 55645 * 28 * 773 * 17 + 45 + 8$ , l'opérateur d'addition est utilisé à 3 reprises. □

Pour apporter plus de clarté, nous allons maintenant donner quelques exemples qui montrent comment vérifier, en appliquant des règles de cette grammaire, que des expressions sont syntaxiquement correctes ou non.

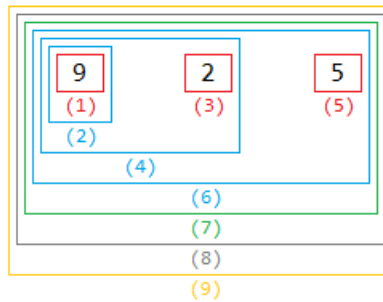
**Ex 1.** Vérifier que le nombre 925 est correctement formé.

Application des règles	Arbre syntaxique
<p>925</p> <p><math>\langle \text{chiffre} \rangle 25</math> (<math>\langle \text{chiffre} \rangle \rightarrow 9</math>) (1)</p> <p><math>\langle \text{nombre} \rangle 25</math> (<math>\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle</math>) (2)</p> <p><math>\langle \text{nombre} \rangle \langle \text{chiffre} \rangle 5</math> (<math>\langle \text{chiffre} \rangle \rightarrow 2</math>) (3)</p> <p><math>\langle \text{nombre} \rangle 5</math> (<math>\langle \text{nombre} \rangle \rightarrow \langle \text{nombre} \rangle \langle \text{chiffre} \rangle</math>) (4)</p> <p><math>\langle \text{nombre} \rangle \langle \text{chiffre} \rangle</math> (<math>\langle \text{chiffre} \rangle \rightarrow 5</math>) (5)</p> <p><math>\langle \text{nombre} \rangle</math> (<math>\langle \text{nombre} \rangle \rightarrow \langle \text{nombre} \rangle \langle \text{chiffre} \rangle</math>) (6)</p> <p><math>\langle \text{exprfois} \rangle</math> (<math>\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle</math>) (7)</p> <p><math>\langle \text{exprplus} \rangle</math> (<math>\langle \text{exprplus} \rangle \rightarrow \langle \text{exprfois} \rangle</math>) (8)</p> <p><math>\langle \text{prog} \rangle</math> (<math>\langle \text{prog} \rangle \rightarrow \langle \text{exprplus} \rangle</math>) (9)</p>	<pre> graph TD     prog["&lt;prog&gt; (9)"] --&gt; exprplus["&lt;exprplus&gt; (8)"]     exprplus --&gt; exprfois["&lt;exprfois&gt; (7)"]     exprfois --&gt; nombre6["&lt;nombre&gt; (6)"]     nombre6 --&gt; nombre4["&lt;nombre&gt; (4)"]     nombre6 --&gt; chiffre5["&lt;chiffre&gt; (5)"]     chiffre5 --&gt; 5["5"]     nombre4 --&gt; nombre2["&lt;nombre&gt; (2)"]     nombre4 --&gt; chiffre3["&lt;chiffre&gt; (3)"]     chiffre3 --&gt; 2["2"]     nombre2 --&gt; chiffre1["&lt;chiffre&gt; (1)"]     chiffre1 --&gt; 9["9"] </pre>

Pour vérifier que la chaîne "925" est un nombre correct en langage Z, nous allons appliquer plusieurs règles de la grammaire. Le but est d'arriver à réduire l'ensemble de la chaîne "925" en le symbole non-terminal le plus abstrait de la grammaire qui est le symbole  $\langle \text{prog} \rangle$ .

Quand on regarde dans la colonne de gauche, on voit que pour passer d'une ligne à la suivante, une règle spécifique de la grammaire est appliquée. Appliquer une règle revient à réduire (remplacer) la partie droite d'une règle par le symbole non-terminal en tête de cette règle. Par exemple, pour passer de la ligne (4) à la ligne (5), c'est-à-dire pour passer de  $\langle \text{nombre} \rangle 5$  à  $\langle \text{nombre} \rangle \langle \text{chiffre} \rangle$ , la règle  $\langle \text{chiffre} \rangle \rightarrow 5$  est appliquée (cette règle se lit "5 est un chiffre"). On remplace le symbole terminal 5 par le symbole non-terminal  $\langle \text{chiffre} \rangle$ .

Voici une autre représentation de l'ordre dans lequel les réductions sont réalisées :



On avance caractère par caractère dans la chaîne en commençant par le symbole terminal le plus à gauche. 9 est réduit en  $\langle \text{chiffre} \rangle$  (1),  $\langle \text{chiffre} \rangle$  est réduit en  $\langle \text{nombre} \rangle$  (2), 2 est réduit en  $\langle \text{chiffre} \rangle$  (3),  $\langle \text{nombre} \rangle \langle \text{chiffre} \rangle$  est réduit en  $\langle \text{nombre} \rangle$  (4), 5 est réduit en  $\langle \text{chiffre} \rangle$  (5),  $\langle \text{nombre} \rangle \langle \text{chiffre} \rangle$  est réduit en  $\langle \text{nombre} \rangle$  (6),  $\langle \text{nombre} \rangle$  est réduit en  $\langle \text{exprfois} \rangle$  (7),  $\langle \text{exprfois} \rangle$  est réduit en  $\langle \text{exprplus} \rangle$  (8),  $\langle \text{exprplus} \rangle$  est réduit en  $\langle \text{prog} \rangle$  (9). On voit la présence de la récursivité, car la règle  $\langle \text{nombre} \rangle \rightarrow \langle \text{nombre} \rangle \langle \text{chiffre} \rangle$  a été appliquée à 2 reprises.

Lors de l'analyse, tant qu'on peut appliquer des règles situées à un niveau de la grammaire, on reste à ce niveau. On ne passe à un niveau de règles plus abstrait que lorsqu'il n'existe plus aucune règle au niveau courant qui peut être appliquée. Ceci explique pourquoi on reste sur les règles ayant pour têtes  $\langle \text{chiffre} \rangle$  et  $\langle \text{nombre} \rangle$  avant, à la fin, de réduire en  $\langle \text{exprfois} \rangle$ , en  $\langle \text{exprplus} \rangle$ , puis en  $\langle \text{prog} \rangle$ .  $\square$

**Ex 2.** Vérifier que l'expression  $52 * 3$  est correctement formée.

Application des règles	Arbre syntaxique
$52 * 3$ $\langle \text{chiffre} \rangle 2 * 3$ ( $\langle \text{chiffre} \rangle \rightarrow 5$ ) (1) $\langle \text{nombre} \rangle 2 * 3$ ( $\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle$ ) (2) $\langle \text{nombre} \rangle \langle \text{chiffre} \rangle * 3$ ( $\langle \text{chiffre} \rangle \rightarrow 2$ ) (3) $\langle \text{nombre} \rangle * 3$ ( $\langle \text{nombre} \rangle \rightarrow \langle \text{nombre} \rangle \langle \text{chiffre} \rangle$ ) (4) $\langle \text{exprfois} \rangle * 3$ ( $\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle$ ) (5) $\langle \text{exprfois} \rangle * \langle \text{chiffre} \rangle$ ( $\langle \text{chiffre} \rangle \rightarrow 3$ ) (6) $\langle \text{exprfois} \rangle * \langle \text{nombre} \rangle$ ( $\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle$ ) (7) $\langle \text{exprfois} \rangle$ ( $\langle \text{exprfois} \rangle \rightarrow \langle \text{exprfois} \rangle * \langle \text{nombre} \rangle$ ) (8) $\langle \text{exprplus} \rangle$ ( $\langle \text{exprplus} \rangle \rightarrow \langle \text{exprfois} \rangle$ ) (9) $\langle \text{prog} \rangle$ ( $\langle \text{prog} \rangle \rightarrow \langle \text{exprplus} \rangle$ ) (10)	

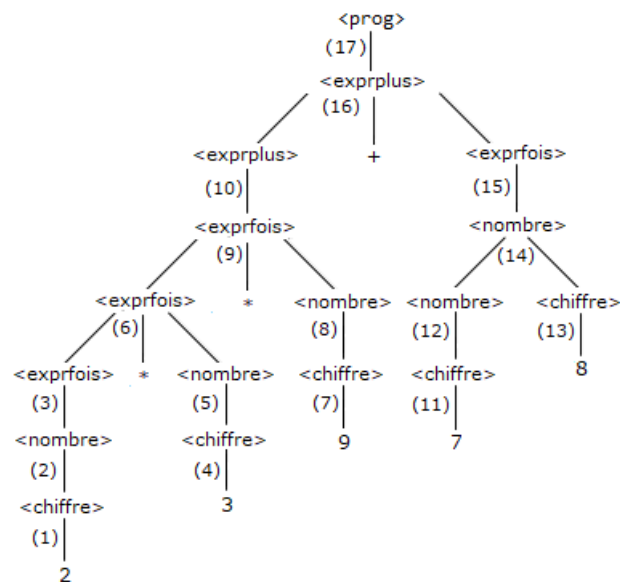
$\square$

**Ex 3.** Vérifier que l'expression  $2 * 3 * 9 + 78$  est correctement formée.

Application des règles :

$2 * 3 * 9 + 78$	
$\langle \text{chiffre} \rangle * 3 * 9 + 78$	$(\langle \text{chiffre} \rangle \rightarrow 2) (1)$
$\langle \text{nombre} \rangle * 3 * 9 + 78$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (2)$
$\langle \text{exprfois} \rangle * 3 * 9 + 78$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle) (3)$
$\langle \text{exprfois} \rangle * \langle \text{chiffre} \rangle * 9 + 78$	$(\langle \text{chiffre} \rangle \rightarrow 3) (4)$
$\langle \text{exprfois} \rangle * \langle \text{nombre} \rangle * 9 + 78$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (5)$
$\langle \text{exprfois} \rangle * 9 + 78$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{exprfois} \rangle * \langle \text{nombre} \rangle) (6)$
$\langle \text{exprfois} \rangle * \langle \text{chiffre} \rangle + 78$	$(\langle \text{chiffre} \rangle \rightarrow 9) (7)$
$\langle \text{exprfois} \rangle * \langle \text{nombre} \rangle + 78$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (8)$
$\langle \text{exprfois} \rangle + 78$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{exprfois} \rangle * \langle \text{nombre} \rangle) (9)$
$\langle \text{exprplus} \rangle + 78$	$(\langle \text{exprplus} \rangle \rightarrow \langle \text{exprfois} \rangle) (10)$
$\langle \text{exprplus} \rangle + \langle \text{chiffre} \rangle 8$	$(\langle \text{chiffre} \rangle \rightarrow 7) (11)$
$\langle \text{exprplus} \rangle + \langle \text{nombre} \rangle 8$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (12)$
$\langle \text{exprplus} \rangle + \langle \text{nombre} \rangle \langle \text{chiffre} \rangle$	$(\langle \text{chiffre} \rangle \rightarrow 8) (13)$
$\langle \text{exprplus} \rangle + \langle \text{nombre} \rangle$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{nombre} \rangle \langle \text{chiffre} \rangle) (14)$
$\langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle) (15)$
$\langle \text{exprplus} \rangle$	$(\langle \text{exprplus} \rangle \rightarrow \langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle) (16)$
$\langle \text{prog} \rangle$	$(\langle \text{prog} \rangle \rightarrow \langle \text{exprplus} \rangle) (17)$

Arbre syntaxique :



□

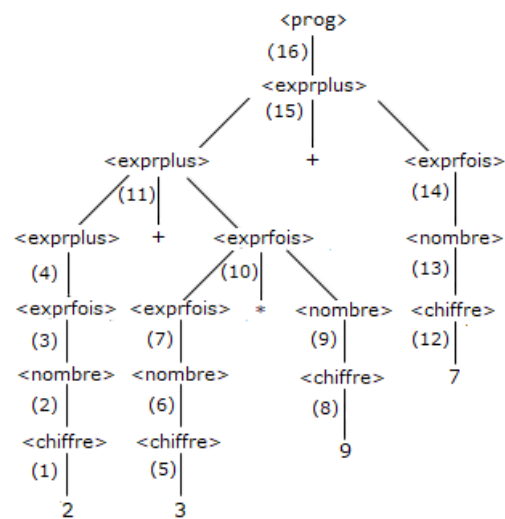


**Ex 4.** Vérifier que l'expression  $2 + 3 * 9 + 78$  est correctement formée.

Application des règles :

$2 + 3 * 9 + 7$	
$\langle \text{chiffre} \rangle + 3 * 9 + 7$	$(\langle \text{chiffre} \rangle \rightarrow 2) (1)$
$\langle \text{nombre} \rangle + 3 * 9 + 7$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (2)$
$\langle \text{exprfois} \rangle + 3 * 9 + 7$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle) (3)$
$\langle \text{exprplus} \rangle + 3 * 9 + 7$	$(\langle \text{exprplus} \rangle \rightarrow \langle \text{exprfois} \rangle) (4)$
$\langle \text{exprplus} \rangle + \langle \text{chiffre} \rangle * 9 + 7$	$(\langle \text{chiffre} \rangle \rightarrow 3) (5)$
$\langle \text{exprplus} \rangle + \langle \text{nombre} \rangle * 9 + 7$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (6)$
$\langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle * 9 + 7$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle) (7)$
$\langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle * \langle \text{chiffre} \rangle + 7$	$(\langle \text{chiffre} \rangle \rightarrow 9) (8)$
$\langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle * \langle \text{nombre} \rangle + 7$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (9)$
$\langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle + 7$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{exprfois} \rangle * \langle \text{nombre} \rangle) (10)$
$\langle \text{exprplus} \rangle + 7$	$(\langle \text{exprplus} \rangle \rightarrow \langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle) (11)$
$\langle \text{exprplus} \rangle + \langle \text{chiffre} \rangle$	$(\langle \text{chiffre} \rangle \rightarrow 7) (12)$
$\langle \text{exprplus} \rangle + \langle \text{nombre} \rangle$	$(\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle) (13)$
$\langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle$	$(\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle) (14)$
$\langle \text{exprplus} \rangle$	$(\langle \text{exprplus} \rangle \rightarrow \langle \text{exprplus} \rangle + \langle \text{exprfois} \rangle) (15)$
$\langle \text{prog} \rangle$	$(\langle \text{prog} \rangle \rightarrow \langle \text{exprplus} \rangle) (16)$

Arbre syntaxique :



□

**Ex 5.** Vérifier que l'expression  $12 * ab$  est correctement formée.

Application des règles	Arbre syntaxique
$12 * ab$ $\langle \text{chiffre} \rangle 2 * ab$ ( $\langle \text{chiffre} \rangle \rightarrow 1$ ) (1) $\langle \text{nombre} \rangle 2 * ab$ ( $\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle$ ) (2) $\langle \text{nombre} \rangle \langle \text{chiffre} \rangle * ab$ ( $\langle \text{chiffre} \rangle \rightarrow 2$ ) (3) $\langle \text{nombre} \rangle * ab$ ( $\langle \text{nombre} \rangle \rightarrow \langle \text{nombre} \rangle \langle \text{chiffre} \rangle$ ) (4) $\langle \text{exprfois} \rangle * ab$ ( $\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle$ ) (5)	

Le symbole terminal  $ab$  ne figure dans aucune règle de la grammaire et donc on ne sait pas ce qu'est ce symbole. Comme il n'a pas été possible de remonter jusqu'au symbole non-terminal  $\langle \text{prog} \rangle$ , l'expression  $12 * ab$  n'est donc pas formée correctement.  $\square$

## 2.2.2 Passage d'une grammaire à un analyseur syntaxique

Écrire une grammaire correcte pour un langage de programmation demande de la réflexion. Cependant, quand la grammaire est correctement écrite, construire à partir de celle-ci le programme appelé analyseur syntaxique qui va servir à déterminer si un programme écrit dans ce langage de programmation respecte ou non la grammaire est une tâche relativement simple.

Voici les principes généraux de conversion de différents types de règles de grammaire en les fonctions correspondantes en Python :

- Un symbole non-terminal  $\langle X \rangle$  figurant en tête d'une règle de grammaire (à gauche de la flèche) entraîne la création, en Python, d'une nouvelle fonction appelée  $X()$ .

```
def X():
    ...
```

- Voici comment convertir en Python des règles de grammaire simples :
  - Pour une règle du type  $\langle X \rangle \rightarrow \langle Y \rangle$ , la fonction  $X()$  invoque la fonction  $Y()$ .

```
def X():
    Y()
```

$\langle X \rangle \rightarrow \langle Y \rangle$

- Pour une règle du type  $\langle X \rangle \rightarrow \langle Y \rangle \text{ a } \langle Z \rangle$ , la fonction  $X()$  invoque la fonction  $Y()$ , puis elle teste si le caractère courant dans la chaîne puis elle invoque la fonction  $Z()$ .

```
def X():
    Y()
    Z()
```

$\langle X \rangle \rightarrow \langle Y \rangle \text{ a } \langle Z \rangle$

- Pour une règle du type  $\langle X \rangle \rightarrow \langle Y \rangle a \langle Z \rangle$  (exemple concret :  $\langle \text{instr} \rangle \rightarrow \langle \text{reg32} \rangle = \langle \text{exprpm} \rangle$ ), la fonction  $X()$  invoque la fonction  $Y()$ , puis elle teste si le caractère courant dans la chaîne analysée est le symbole terminal  $a$ . Si c'est le cas, alors elle invoque la fonction  $Z()$ . **La présence d'un symbole terminal dans une règle de grammaire entraîne l'usage d'une alternative "if" dans la fonction en Python.**

```
def X():
    Y()
    if SymboleCourant(1) == 'a':
        SymboleSuivant(1)
        Z()
```

Le code des fonctions `SymboleCourant()` et `SymboleSuivant()` sera donné plus loin. En quelques mots, `SymboleCourant(1)` retourne le caractère courant dans la chaîne analysée et `SymboleSuivant(1)` passe au caractère suivant dans la chaîne analysée. La fonction `SymboleSuivant()` est invoquée dès qu'est établie la correspondance entre le symbole terminal et le caractère courant dans la chaîne analysée.

- Pour une règle du type  $\langle X \rangle \rightarrow a \langle Y \rangle b$  (exemple concret :  $\langle \text{prog} \rangle \rightarrow \text{start} \langle \text{exprplus} \rangle \text{stop}$ ), la fonction  $X()$  teste si le caractère courant dans la chaîne analysée est le symbole terminal  $a$ . Si c'est le cas, alors elle invoque la fonction  $Y()$ . Ensuite, elle teste si le caractère courant dans la chaîne analysée est le symbole terminal  $b$ .

```
def X():
    if SymboleCourant(1) == 'a':
        SymboleSuivant(1)
        Y()
    if SymboleCourant(1) == 'b':
        SymboleSuivant(1)
```

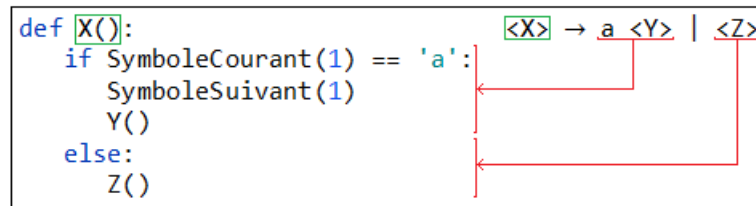
- Voici comment convertir en Python plusieurs règles ayant pour tête le même symbole non-terminal :

- Pour des règles du type  $\langle X \rangle \rightarrow a \mid b \mid c$  (exemple concret :  $\langle \text{chiffre} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$ ), la fonction  $X()$  teste si le caractère courant dans la chaîne analysée est le symbole terminal  $a$ ,  $b$  ou  $c$ . Si c'est le cas, alors elle retourne VRAI, sinon elle retourne FAUX.

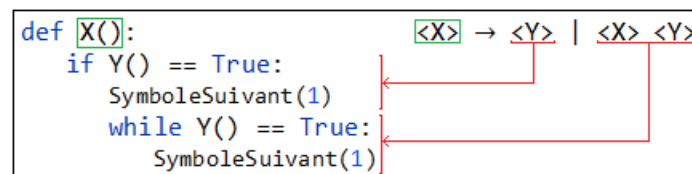
```
def X():
    if SymboleCourant(1) in "abc":
        SymboleSuivant(1)
        return True
    else:
        return False
```

- Pour des règles du type  $\langle X \rangle \rightarrow a \langle Y \rangle \mid \langle Z \rangle$  (exemple concret :  $\langle \text{facteur} \rangle \rightarrow (\langle \text{exprplus} \rangle) \mid \langle \text{nombre} \rangle$ ), la fonction  $X()$  teste si le caractère courant dans la chaîne analysée est le symbole terminal  $a$ . Si c'est le cas, alors elle invoque la fonction  $Y()$ , sinon elle invoque la fonction  $Z()$ . C'est la présence du symbole terminal  $a$  au début du corps de la règle  $\langle X \rangle \rightarrow a \langle Y \rangle$  qui permet, à un moment donné, de choisir entre cette

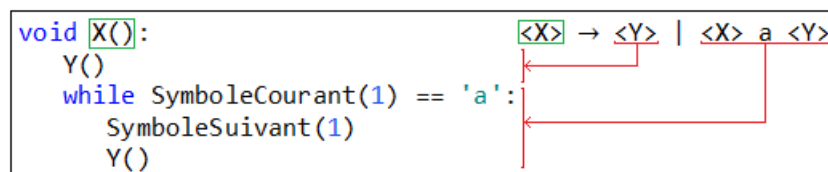
règle de grammaire et la règle  $\langle X \rangle \rightarrow \langle Z \rangle$ .



- Voici comment convertir en Python des règles ayant pour tête le même symbole non-terminal qui incorporent de la **récurtivité**. La récursivité est utilisée dans une grammaire pour pouvoir répéter successivement un même motif dans la chaîne analysée. **La récursivité dans une règle de grammaire entraîne l'usage d'une boucle "while" dans la fonction en Python.**
  - Pour des règles du type  $\langle X \rangle \rightarrow \langle Y \rangle \mid \langle X \rangle \langle Y \rangle$  (exemple concret :  $\langle \text{nombre} \rangle \rightarrow \langle \text{chiffre} \rangle \mid \langle \text{nombre} \rangle \langle \text{chiffre} \rangle$ ), la fonction  $X()$  invoque une première fois la fonction  $Y()$ . Si la fonction  $Y()$  retourne VRAI, alors la fonction  $X()$  invoque la fonction  $Y()$  tant que celle-ci retourne VRAI.



- Pour des règles du type  $\langle X \rangle \rightarrow \langle Y \rangle \mid \langle X \rangle a \langle Y \rangle$  (exemple concret :  $\langle \text{exprfois} \rangle \rightarrow \langle \text{nombre} \rangle \mid \langle \text{exprfois} \rangle * \langle \text{nombre} \rangle$ ), la fonction  $X()$  invoque une première fois la fonction  $Y()$ . Puis, tant que le caractère courant dans la chaîne analysée est le symbole terminal  $a$ , la fonction  $X()$  invoque la fonction  $Y()$ .



## 2.2.3 Analyseur syntaxique

Reprenons la grammaire donnée dans la section 2.2.1.

```
<prog>      -> start <exprplus> stop
<exprplus> -> <exprfois> | <exprplus> + <exprfois>
<exprfois> -> <nombre> | <exprfois> * <nombre>
<nombre>   -> <chiffre> | <nombre> <chiffre>
<chiffre>  -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Les symboles terminaux *start* et *stop* indiquent le début et la fin du programme source en langage Z.

Voici l'analyseur syntaxique construit à partir de cette grammaire :

```
# -----
# <prog> -> start <exprplus> stop
# -----

def Prog():
    SymboleSuivant(0) # supprimer les espaces éventuels au début de la chaîne à analyser
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        ExprPlus()
        if SymboleCourant(4) == "stop":
            SymboleSuivant(4)
            return True
        else:
            return False
    else:
        return False

# -----
# <exprplus> -> <exprfois> | <exprplus> + <exprfois>
# -----

def ExprPlus():
    ExprFois()
    while SymboleCourant(1) == '+':
        SymboleSuivant(1)
        ExprFois()

# -----
# <exprfois> -> <nombre> | <exprfois> * <nombre>
# -----

def ExprFois():
    Nombre()
    while SymboleCourant(1) == '*':
        SymboleSuivant(1)
        Nombre()

# -----
# <nombre> -> <chiffre> | <nombre><chiffre>
# -----

def Nombre():
    if Chiffre() == True:
        SymboleSuivant(1)
        while Chiffre() == True:
            SymboleSuivant(1)

# -----
# <chiffre> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":
        return True
    else:
        return False

# -----
# Fonction qui retourne les n caractères courants dans programme
# -----

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]
```

```

# -----
# Fonction qui va n caractères plus loin dans programme
# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----
# code principal
# -----

programme = ("start"
             "2 + 337 * 2 + 645"
             "stop")
posCourante = 0

if Prog() == True:
    print("\nAnalyse syntaxique terminée avec succès!")
else:
    print("Erreur : le caractere", SymboleCourant(1),
          "à la " + str(posCourante + 1) + "e position est invalide!")

```

Commentaires :

- On voit que chaque symbole non-terminal situé en tête de règle est devenu une fonction dans l'analyseur syntaxique.

<i>Symboles non-terminaux</i>	<i>Fonctions en Python</i>
<prog>	Prog()
<exprplus>	ExprPlus()
<exprfois>	ExprFois()
<nombre>	Nombre()
<chiffre>	Chiffre()

- On voit que la récursivité dans une règle de grammaire devient une boucle dans la fonction correspondante de l'analyseur syntaxique. Pour rappel, on a de la récursivité dans une règle de grammaire lorsque le même symbole non-terminal est présent en tête et dans le corps de cette règle. La récursivité rend possible la répétition un nombre quelconque de fois d'un motif similaire.
- Le code source en langage Z à analyser se trouve dans la chaîne de caractères *programme*.

```

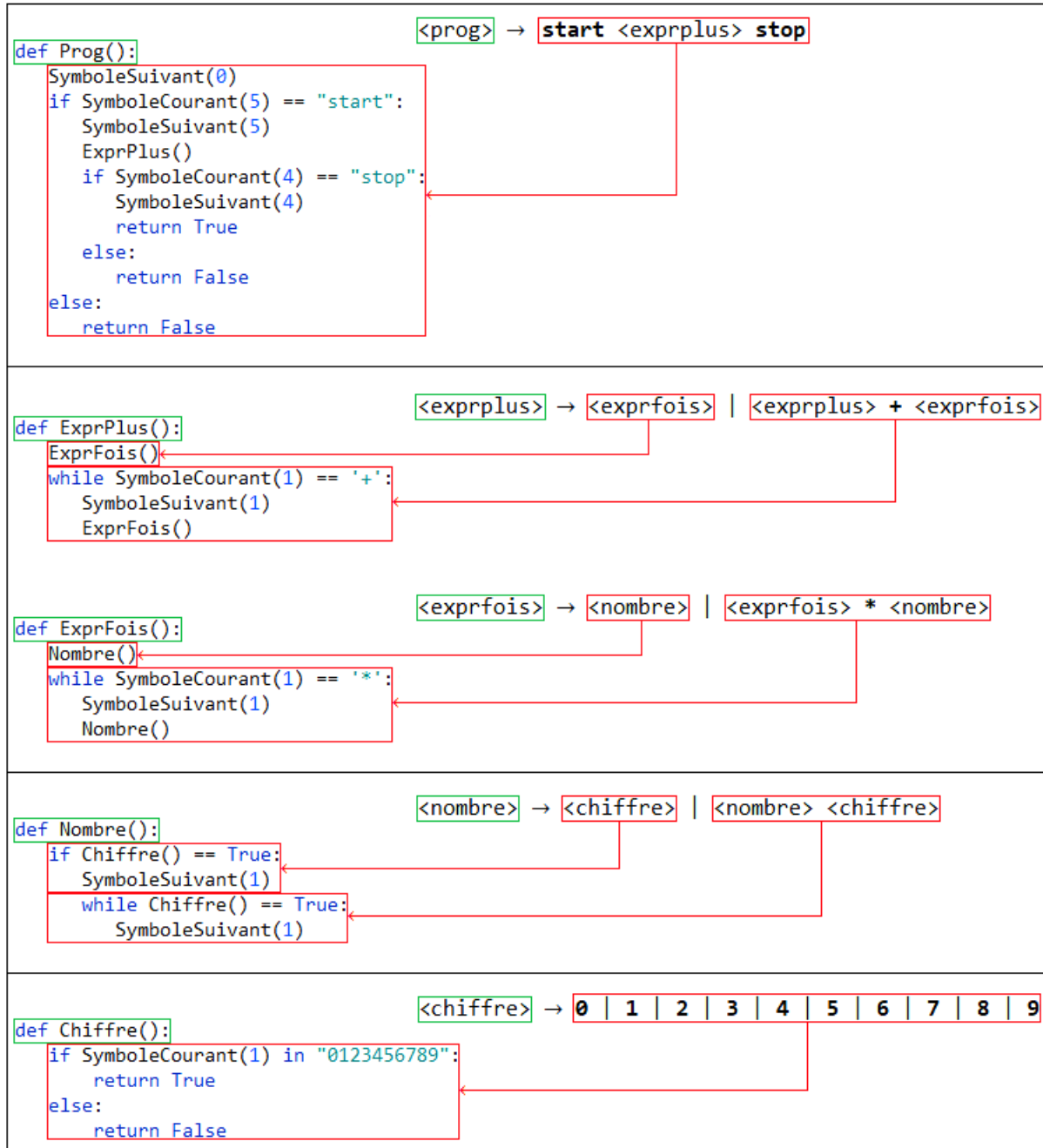
programme = ("start"
             "2 + 337 * 2 + 645"
             "stop")

```

- La variable *posCourante* mémorise la position du caractère courant dans la chaîne de caractères *programme*.
- Les fonctions *SymboleCourant(n)* et *SymboleSuivant(n)* simplifient les accès aux caractères de la chaîne à analyser. *SymboleCourant(n)* retourne les *n* caractères courants dans la chaîne de caractères *programme* et *SymboleSuivant(n)* permet d'aller *n* caractères plus loin

dans la chaîne de caractères *programme*.

Voici une illustration de la conversion des règles de la grammaire en les fonctions de l'analyseur syntaxique :



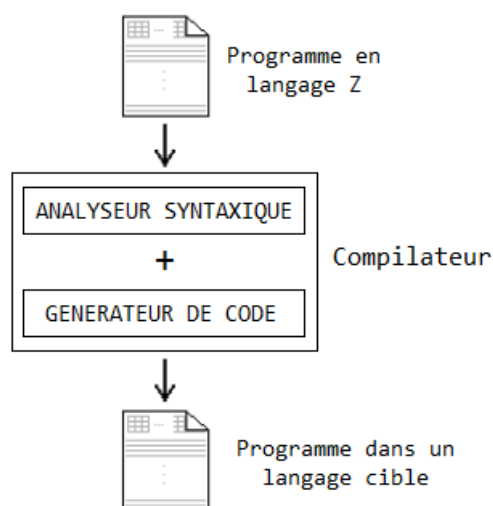
## Chapitre 3

# Principes de conception d'un compilateur

### 3.1 Introduction

Pour réaliser un compilateur pour un langage que nous nommerons *langage Z*, il faut passer au minimum par les étapes suivantes :

1. Avoir une idée précise du langage de programmation Z souhaité pour pouvoir rédiger la **grammaire** du langage Z.
2. Sur base de cette grammaire, écrire un programme appelé **analyseur syntaxique**. Celui-ci a pour but de déterminer si un programme écrit en langage Z respecte la grammaire de ce langage ou non pour savoir s'il est correctement écrit en langage Z ou non.
3. Ajouter dans l'analyseur syntaxique les instructions de génération du code cible pour former le **compilateur** Z. À partir d'un programme écrit en langage Z, le compilateur génère un programme équivalent, mais exprimé dans un langage cible de plus bas niveau (exemple : l'assembleur).





## 3.2 Première version du langage Z

### 3.2.1 Grammaire

```

<prog>      → start <exprplus> stop
<exprplus> → <exprfois> | <exprplus> + <exprfois>
<exprfois> → <nombre> | <exprfois> * <nombre>
<nombre>   → <chiffre> | <nombre> <chiffre>
<chiffre>  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

### 3.2.2 Compilateur

Dans la section 2.2.2, un analyseur syntaxique a déjà été créé à partir de la grammaire du langage Z. Nous allons intégrer dans cet analyseur syntaxique les instructions de génération de code pour former le compilateur Z :

```

def Prog():
    SymboleSuivant(0)
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        codeCible.append("void main()")
        codeCible.append("{")
        codeCible.append("\t_asm")
        codeCible.append("\t{")
        ExprPlus()
        if SymboleCourant(4) == "stop":
            SymboleSuivant(4)
            codeCible.append("\t\tpop eax")
            codeCible.append("\t}")
            codeCible.append("}")
            return True
        else:
            return False
    else:
        return False

# -----

def ExprPlus():
    ExprFois()
    while SymboleCourant(1) == '+':
        SymboleSuivant(1)
        ExprFois()
        codeCible.append("\t\tpop ebx")
        codeCible.append("\t\tpop eax")
        codeCible.append("\t\tadd eax, ebx")
        codeCible.append("\t\tpush eax")

# -----

def ExprFois():
    Nombre()
    while SymboleCourant(1) == '*':
        SymboleSuivant(1)
        Nombre()
        codeCible.append("\t\tpop ebx")
        codeCible.append("\t\tpop eax")
        codeCible.append("\t\timul eax, ebx")
        codeCible.append("\t\tpush eax")

```

```

# -----
def Nombre():
    if Chiffre() == True:
        nb = ord(SymboleCourant(1)) - 0x30
        SymboleSuivant(1)
        while Chiffre() == True:
            nb = nb * 10 + ord(SymboleCourant(1)) - 0x30
            SymboleSuivant(1)
        codeCible.append("\t\tpush dword ptr " + str(nb))

# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":
        return True
    else:
        return False

# -----

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]

# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----

programme = ("start"
             "5 + 337 * 2 + 645"
             "stop")
codeCible = []
posCourante = 0

if Prog() == True:
    for c in codeCible:
        print(c)
    print("\nCompilation terminée avec succès!")
else:
    print("Erreur de compilation : le caractere", SymboleCourant(1),
          "à la " + str(posCourante + 1) + "e position est invalide!")

```

Voici le résultat d'une compilation réalisée avec ce compilateur :

<i>Programme source en langage Z</i>	<i>Programme cible en assembleur pour Visual Studio</i>
<pre> start   5 + 337 * 2 + 645 stop </pre>	<pre> void main() {     _asm     {         push dword ptr 5         push dword ptr 337         push dword ptr 2         pop ebx         pop eax         imul eax, ebx         push eax     } } </pre>

	<pre> pop ebx pop eax add eax, ebx push eax push dword ptr 645 pop ebx pop eax add eax, ebx push eax pop eax     } }</pre>
--	--

Quelques commentaires :

- Bien que le code en assembleur généré par notre compilateur ne soit pas optimisé et utilise abondamment la pile, son exécution donne le résultat attendu. Ainsi, quand on l'exécute dans Visual Studio, on obtient dans EAX le résultat attendu (la valeur 1324).
- La liste *codeCible* sert à stocker les lignes de code en assembleur générées par le compilateur.
- Quand on regarde le code en assembleur généré par ce compilateur, on peut dire que :
  - Chaque nombre est empilé (exemple : push dword ptr 337).
  - Deux nombres sont dépilés et placés dans EBX et EAX juste avant réalisation de toute opération d'addition ou de multiplication. L'opération est ensuite appliquée sur ces 2 nombres et le résultat est empilé.
  - Le résultat final est dépilé et placé dans EAX.
  - Comme la multiplication est plus prioritaire que l'addition, imul eax, ebx apparaît avant add eax, ebx.

## 3.3 Deuxième version du langage Z

### 3.3.1 Grammaire

Voici la grammaire déjà donnée dans la section 3.2.1 :

```

<prog>      → start <exprplus> stop
<exprplus> → <exprfois> | <exprplus> + <exprfois>
<exprfois> → <nombre> | <exprfois> * <nombre>
<nombre>   → <chiffre> | <nombre> <chiffre>
<chiffre>  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Voici une version étendue de cette grammaire :

```

<prog>    → start <exprpm> stop
<exprpm>  → <exprfd> | <exprpm> + <exprfd> | <exprpm> - <exprfd>
<exprfd>  → <facteur> | <exprfd> * <facteur> | <exprfd> / <facteur>
<facteur> → ( <exprpm> ) | <nombre>
<nombre>  → <chiffre> | <nombre> <chiffre>
<chiffre> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Dans cette nouvelle version de la grammaire :

- La soustraction a été ajoutée en veillant à lui donner la même priorité que l'addition. Le symbole non-terminal <exprplus> a été renommé en <exprpm> ("expression plus moins").
- La division a été ajoutée en veillant à lui donner la même priorité que la multiplication. Le symbole non-terminal <exprfois> a été renommé en <exprfd> ("expression fois divisé").
- Les parenthèses ont été ajoutées. Une expression entre parenthèses est prise en compte au même niveau qu'un nombre, car ils ont <facteur> comme tête de règle. Le facteur est l'élément de base sur lequel un opérateur peut être appliqué. Un opérateur peut maintenant être appliqué autant sur un nombre que sur une expression entre parenthèses. Dans une expression entre parenthèses, on doit pouvoir utiliser les opérateurs +, -, \* et /. Ceci explique pourquoi on trouve <exprpm> dans la règle de grammaire <facteur> → (<exprpm>).

### 3.3.2 Compilateur

Voici le compilateur construit à partir de la nouvelle version de la grammaire :

```

def Prog():
    SymboleSuivant(0)
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        codeCible.append("void main()")
        codeCible.append("{")
        codeCible.append("\t_asm")
        codeCible.append("\t{")
        ExprPM()
        if SymboleCourant(4) == "stop":
            SymboleSuivant(4)
            codeCible.append("\t\tpop eax")
            codeCible.append("\t}")
            codeCible.append("}")
            return True
        else:
            return False
    else:
        return False

# -----

def ExprPM():
    ExprFD()
    while SymboleCourant(1) in "+-":
        if SymboleCourant(1) == '+':
            SymboleSuivant(1)

```

```

ExprFD()
codeCible.append("\t\tpop ebx")
codeCible.append("\t\tpop eax")
codeCible.append("\t\tadd eax, ebx")
codeCible.append("\t\tpush eax")
elif SymboleCourant(1) == '-':
    SymboleSuivant(1)
    ExprFD()
    codeCible.append("\t\tpop ebx")
    codeCible.append("\t\tpop eax")
    codeCible.append("\t\tsub eax, ebx")
    codeCible.append("\t\tpush eax")

# -----

def ExprFD():
    Facteur()
    while SymboleCourant(1) in "*/":
        if SymboleCourant(1) == '*':
            SymboleSuivant(1)
            Facteur()
            codeCible.append("\t\tpop ebx")
            codeCible.append("\t\tpop eax")
            codeCible.append("\t\timul eax, ebx")
            codeCible.append("\t\tpush eax")
        elif SymboleCourant(1) == '/':
            SymboleSuivant(1)
            Facteur()
            codeCible.append("\t\tpop ebx")
            codeCible.append("\t\tpop eax")
            codeCible.append("\t\tcdq")
            codeCible.append("\t\tidiv ebx")
            codeCible.append("\t\tpush eax")

# -----

def Facteur():
    if SymboleCourant(1) == '(':
        SymboleSuivant(1)
        ExprPM()
        if SymboleCourant(1) == ')':
            SymboleSuivant(1)
    else:
        Nombre()

# -----

def Nombre():
    if Chiffre() == True:
        nb = ord(SymboleCourant(1)) - 0x30
        SymboleSuivant(1)
        while Chiffre() == True:
            nb = nb * 10 + ord(SymboleCourant(1)) - 0x30
            SymboleSuivant(1)
        codeCible.append("\t\tpush dword ptr " + str(nb))

# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":
        return True
    else:
        return False

# -----

```

```

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]

# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----

programme = ("start"
             "512 * (144 + 45412) * 12 - 541 / 2"
             "stop")
codeCible = []
posCourante = 0

if Prog() == True:
    for c in codeCible:
        print(c)
    print("\nCompilation terminée avec succès!")
else:
    print("Erreur de compilation : le caractere", SymboleCourant(1),
          "à la " + str(posCourante + 1) + "e position est invalide!")

```

Voici le résultat d'une compilation réalisée avec ce compilateur :

<i>Programme source en langage Z</i>	<i>Programme cible en assembleur pour Visual Studio</i>
<pre> start   512 * (144 + 45412) * 12 - 541 / 2 stop </pre>	<pre> void main() {     _asm     {         push dword ptr 512         push dword ptr 144         push dword ptr 45412         pop ebx         pop eax         add eax, ebx         push eax         pop ebx         pop eax         imul eax, ebx         push eax         push dword ptr 12         pop ebx         pop eax         imul eax, ebx         push eax         push dword ptr 541         push dword ptr 2         pop ebx         pop eax         cdq         idiv ebx         push eax         pop ebx         pop eax         sub eax, ebx         push eax </pre>

	pop eax
}	
}	

## 3.4 Troisième version du langage Z

### 3.4.1 Grammaire

**Ex.** Voici la grammaire donnée déjà dans la section 3.3.1 :

```

<prog>    → start <exprpm> stop
<exprpm>  → <exprfd> | <exprpm> + <exprfd> | <exprpm> - <exprfd>
<exprfd>  → <facteur> | <exprfd> * <facteur> | <exprfd> / <facteur>
<facteur> → ( <exprpm> ) | <nombre>
<nombre>  → <chiffre> | <nombre> <chiffre>
<chiffre> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Voici une version étendue de cette grammaire :

```

<prog>      → start <suiteinstr> stop
<suiteinstr> → <instr> | <suiteinstr> ; <instr>
<instr>      → <reg32> = <exprpm>
<exprpm>     → <exprfd> | <exprpm> + <exprfd> | <exprpm> - <exprfd>
<exprfd>     → <facteur> | <exprfd> * <facteur> | <exprfd> / <facteur>
<facteur>    → ( <exprpm> ) | <nombre> | <reg32>
<nombre>     → <chiffre> | <nombre> <chiffre>
<chiffre>    → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<reg32>      → eax | ebx | ecx | edx | esi | edi

```

Avec cette nouvelle grammaire :

- L'opérateur d'affectation = permet de transférer dans un registre général de 32 bits le résultat de l'évaluation d'une l'expression. Comme cet opérateur est moins prioritaire que les opérateurs + et -, la règle <instr> → <reg32> = <exprpm> fait référence au symbole non-terminal <exprpm>. Il n'y a aucune récursivité au niveau de l'opérateur d'affectation, car il ne peut être présent qu'une et une seule fois dans une instruction d'affectation.
- Pour pouvoir avoir plusieurs instructions d'affectation, on trouve de la récursivité au niveau du symbole non-terminal <suiteinstr>. Le symbole terminal ; sert de séparateur entre une instruction d'affectation et la suivante.
- Au niveau du <facteur>, on peut avoir maintenant une expression entre parenthèses, un nombre ou un registre général de 32 bits.

Voici une version étendue de cette grammaire :

<code>&lt;prog&gt;</code>	<code>→ start &lt;suiteinstr&gt; stop</code>
<code>&lt;suiteinstr&gt;</code>	<code>→ &lt;instr&gt;   &lt;suiteinstr&gt; ; &lt;instr&gt;</code>
<code>&lt;instr&gt;</code>	<code>→ &lt;reg32&gt; = &lt;exprpm&gt;</code>
<code>&lt;exprpm&gt;</code>	<code>→ &lt;exprfd&gt;   &lt;exprpm&gt; + &lt;exprfd&gt;   &lt;exprpm&gt; - &lt;exprfd&gt;</code>
<code>&lt;exprfd&gt;</code>	<code>→ &lt;facteur&gt;   &lt;exprfd&gt; * &lt;facteur&gt;   &lt;exprfd&gt; / &lt;facteur&gt;</code>
<code>&lt;facteur&gt;</code>	<code>→ ( &lt;exprpm&gt; )   &lt;nombre&gt;   &lt;reg32&gt;</code>
<code>&lt;nombre&gt;</code>	<code>→ &lt;chiffre&gt;   &lt;nombre&gt; &lt;chiffre&gt;</code>
<code>&lt;chiffre&gt;</code>	<code>→ 0   1   2   3   4   5   6   7   8   9</code>
<code>&lt;reg32&gt;</code>	<code>→ eax   ebx   ecx   edx   esi   edi</code>

Avec cette nouvelle grammaire :

- L'opérateur d'affectation = permet de transférer dans un registre général de 32 bits le résultat de l'évaluation d'une l'expression. Comme cet opérateur est moins prioritaire que les opérateurs + et -, la règle `<instr> → <reg32> = <exprpm>` fait référence au symbole non-terminal `<exprpm>`. Il n'y a aucune récursivité au niveau de l'opérateur d'affectation, car il ne peut être présent qu'une et une seule fois dans une instruction d'affectation.
- Pour pouvoir avoir plusieurs instructions d'affectation, on trouve de la récursivité au niveau du symbole non-terminal `<suiteinstr>`. Le symbole terminal ; sert de séparateur entre une instruction d'affectation et la suivante.
- Au niveau du `<facteur>`, on peut avoir maintenant une expression entre parenthèses, un nombre ou un registre général de 32 bits.

### 3.4.2 Compilateur

Voici le compilateur construit à partir de la nouvelle version de la grammaire :

```
def Prog():
    SymboleSuivant(0)
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        codeCible.append("void main()")
        codeCible.append("{")
        codeCible.append("\t_asm")
        codeCible.append("\t{")
        SuiteInstr()
        if SymboleCourant(4) == "stop":
            SymboleSuivant(4)
            codeCible.append("\t}")
            codeCible.append("}")
            return True
        else:
            return False
    else:
        return False

# -----

def SuiteInstr():
    Instr()
    while SymboleCourant(1) == ';':
        SymboleSuivant(1)
```



```

Instr()

# -----

def Instr():
    if Reg32() == True:
        regCible = SymboleCourant(3)
        SymboleSuivant(3)
        if SymboleCourant(1) == '=':
            SymboleSuivant(1)
            ExprPM()
            codeCible.append("\t\tpop " + regCible)

# -----

def ExprPM():
    ExprFD()
    while SymboleCourant(1) in "+-":
        if SymboleCourant(1) == '+':
            SymboleSuivant(1)
            ExprFD()
            codeCible.append("\t\tpop ebx")
            codeCible.append("\t\tpop eax")
            codeCible.append("\t\tadd eax, ebx")
            codeCible.append("\t\tpush eax")
        elif SymboleCourant(1) == '-':
            SymboleSuivant(1)
            ExprFD()
            codeCible.append("\t\tpop ebx")
            codeCible.append("\t\tpop eax")
            codeCible.append("\t\tsub eax, ebx")
            codeCible.append("\t\tpush eax")

# -----

def ExprFD():
    Facteur()
    while SymboleCourant(1) in "*/":
        if SymboleCourant(1) == '*':
            SymboleSuivant(1)
            Facteur()
            codeCible.append("\t\tpop ebx")
            codeCible.append("\t\tpop eax")
            codeCible.append("\t\timul eax, ebx")
            codeCible.append("\t\tpush eax")
        elif SymboleCourant(1) == '/':
            SymboleSuivant(1)
            Facteur()
            codeCible.append("\t\tpop ebx")
            codeCible.append("\t\tpop eax")
            codeCible.append("\t\tcdq")
            codeCible.append("\t\tidiv ebx")
            codeCible.append("\t\tpush eax")

# -----

def Facteur():
    if SymboleCourant(1) == '(':
        SymboleSuivant(1)
        ExprPM()
        if SymboleCourant(1) == ')':
            SymboleSuivant(1)
    elif Reg32() == True:
        codeCible.append("\t\tpush " + SymboleCourant(3))
        SymboleSuivant(3)
    else:

```

```

    Nombre()

# -----

def Nombre():
    if Chiffre() == True:
        nb = ord(SymboleCourant(1)) - 0x30
        SymboleSuivant(1)
        while Chiffre() == True:
            nb = nb * 10 + ord(SymboleCourant(1)) - 0x30
            SymboleSuivant(1)
        codeCible.append("\t\tpush dword ptr " + str(nb))

# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":
        return True
    else:
        return False

# -----

def Reg32():
    if SymboleCourant(3) in ["eax", "ebx", "ecx", "edx", "esi", "edi"]:
        return True
    else:
        return False

# -----

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]

# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----

programme = ("start"
             "ecx = 3;"
             "esi = 45 + 6 / ecx;"
             "ebx = 512 * (144 + 912 + ecx) * 51 - esi / 2"
             "stop")
codeCible = []
posCourante = 0

if Prog() == True:
    for c in codeCible:
        print(c)
    print("\nCompilation terminée avec succès!")
else:
    print("Erreur de compilation : le caractere", SymboleCourant(1),
          "à la " + str(posCourante + 1) + "e position est invalide!")

```

Voici le résultat d'une compilation réalisée avec ce compilateur :

<i>Programme source en langage Z</i>	<i>Programme cible en assembleur pour VS</i>
<pre> start     ecx = 3;     esi = 45 + 6 / ecx;     ebx = 512 * (144 + 912 + ecx) * 51 - esi / 2 stop </pre>	<pre> void main() {     _asm     {         push dword ptr 3         pop ecx         push dword ptr 45         push dword ptr 6         push ecx         pop ebx         pop eax         cdq         idiv ebx         push eax         pop ebx         pop eax         add eax, ebx         push eax         pop esi         push dword ptr 512         push dword ptr 144         push dword ptr 912         pop ebx         pop eax         add eax, ebx         push eax         push ecx         pop ebx         pop eax         add eax, ebx         push eax         pop ebx         pop eax         imul eax, ebx         push eax         push dword ptr 51         pop ebx         pop eax         imul eax, ebx         push eax         push esi         push dword ptr 2         pop ebx         pop eax         cdq         idiv ebx         push eax         pop ebx         pop eax         sub eax, ebx         push eax         pop ebx     } } </pre>

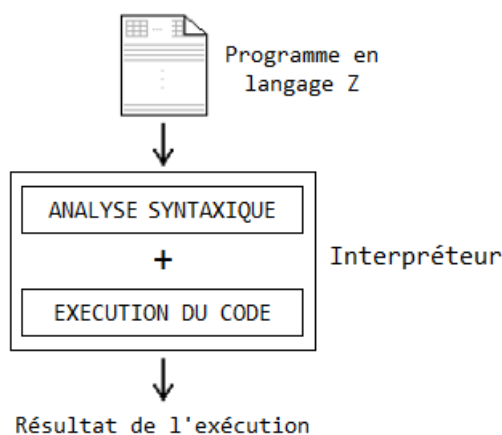
## Chapitre 4

# Principes de conception d'un interpréteur

### 4.1 Introduction

Pour réaliser un interpréteur pour un langage que nous nommerons *langage Z*, il faut passer au minimum par les étapes suivantes :

1. Avoir une idée précise du langage de programmation Z souhaité pour pouvoir rédiger la **grammaire** du langage Z.
2. Sur base de cette grammaire, écrire un programme appelé **analyseur syntaxique**. Celui-ci a pour but de déterminer si un programme écrit en langage Z respecte la grammaire de ce langage ou non pour savoir s'il est correctement écrit en langage Z ou non.
3. Ajouter dans l'analyseur syntaxique les instructions d'exécution du programme en langage Z pour former l'**interpréteur Z**.



## 4.2 Première version du langage Z

### 4.2.1 Grammaire

```

<prog>      → start <exprplus> stop
<exprplus> → <exprfois> | <exprplus> + <exprfois>
<exprfois> → <nombre> | <exprfois> * <nombre>
<nombre>   → <chiffre> | <nombre> <chiffre>
<chiffre>  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

### 4.2.2 Interpréteur

Dans la section 2.2.2, un analyseur syntaxique a déjà été créé à partir de la grammaire du langage Z. Nous allons intégrer dans cet analyseur syntaxique les instructions d'exécution du code pour former l'interpréteur Z :

```

def Prog():
    SymboleSuivant(0)
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        ExprPlus()
        if SymboleCourant(4) == "stop":
            SymboleSuivant(4)
            return True
        else:
            return False
    else:
        return False

# -----

def ExprPlus():
    ExprFois()
    while SymboleCourant(1) == '+':
        SymboleSuivant(1)
        ExprFois()
        nb2 = popResultat()
        nb1 = popResultat()
        resultat.append(nb1 + nb2)

# -----

def ExprFois():
    Nombre()
    while SymboleCourant(1) == '*':
        SymboleSuivant(1)
        Nombre()
        nb2 = popResultat()
        nb1 = popResultat()
        resultat.append(nb1 * nb2)

# -----

def Nombre():
    if Chiffre() == True:
        nb = ord(SymboleCourant(1)) - 0x30
        SymboleSuivant(1)
        while Chiffre() == True:
            nb = nb * 10 + ord(SymboleCourant(1)) - 0x30

```

```

        SymboleSuivant(1)
        resultat.append(nb)

# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":
        return True
    else:
        return False

# -----

def popResultat():
    if len(resultat) > 0:
        return resultat.pop()
    else:
        print("Valeur manquante dans l'instruction !")
        exit()

# -----

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]

# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----

programme = ("start"
             "5 + 337 * 2 + 645"
             "stop")
resultat = []
posCourante = 0

if Prog() == True:
    print("Le résultat de l'expression est :", popResultat())
else:
    print("Erreur dans le code source à partir de la " + str(posCourante + 1) + "e position !")

```

Cet interpréteur affiche la valeur 1324 comme résultat de l'exécution de l'expression  $5 + 337 * 2 + 645$ .

Pour stocker tous les résultats intermédiaires, les instructions d'exécution du code en langage Z à l'intérieur de l'interpréteur exploite la liste *resultat* en tant que pile, c'est-à-dire que les accès y sont faits selon le mécanisme LIFO (Last-In, First-Out, dernier placé, premier retiré). On voit que pour réaliser toute opération d'addition ou de multiplication :

- Chaque nombre est empilé dans *resultat*.
- Deux nombres sont dépilés de *resultat* juste avant la réalisation de l'opération. L'opération est ensuite appliquée sur ces 2 nombres et le résultat est empilé dans *resultat*.
- À la fin, le résultat de l'exécution de toute l'expression est dépilé de *resultat* et est affiché.

## 4.3 Deuxième version du langage Z

### 4.3.1 Grammaire

Voici la grammaire déjà donnée dans la section 4.2.1 :

```
<prog>      → start <exprplus> stop
<exprplus> → <exprfois> | <exprplus> + <exprfois>
<exprfois> → <nombre> | <exprfois> * <nombre>
<nombre>   → <chiffre> | <nombre> <chiffre>
<chiffre>  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Voici une version étendue de cette grammaire :

```
<prog>      → start <exprpm> stop
<exprpm>   → <exprfd> | <exprpm> + <exprfd> | <exprpm> - <exprfd>
<exprfd>   → <facteur> | <exprfd> * <facteur> | <exprfd> / <facteur>
<facteur>  → ( <exprpm> ) | <nombre>
<nombre>   → <chiffre> | <nombre> <chiffre>
<chiffre>  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Dans cette nouvelle version de la grammaire :

- La soustraction a été ajoutée en veillant à lui donner la même priorité que l'addition. Le symbole non-terminal <exprplus> a été renommé en <exprpm> ("expression plus moins").
- La division a été ajoutée en veillant à lui donner la même priorité que la multiplication. Le symbole non-terminal <exprfois> a été renommé en <exprfd> ("expression fois divisé").
- Les parenthèses ont été ajoutées. Une expression entre parenthèses est prise en compte au même niveau qu'un nombre, car ils ont <facteur> comme tête de règle. Le facteur est l'élément de base sur lequel un opérateur peut être appliqué. Un opérateur peut maintenant être appliqué autant sur un nombre que sur une expression entre parenthèses. Dans une expression entre parenthèses, on doit pouvoir utiliser les opérateurs +, -, \* et /. Ceci explique pourquoi on trouve <exprpm> dans la règle de grammaire <facteur> → (<exprpm>).

### 4.3.2 Interpréteur

Voici l'interpréteur construit à partir de la nouvelle version de la grammaire :

```
def Prog():
    SymboleSuivant(0)
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        ExprPM()
    if SymboleCourant(4) == "stop":
        SymboleSuivant(4)
        return True
    else:
        return False
```

```

    else:
        return False

# -----

def ExprPM():
    ExprFD()
    while SymboleCourant(1) in "+-":
        if SymboleCourant(1) == '+':
            SymboleSuivant(1)
            ExprFD()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(nb1 + nb2)
        elif SymboleCourant(1) == '-':
            SymboleSuivant(1)
            ExprFD()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(nb1 - nb2)

# -----

def ExprFD():
    Facteur()
    while SymboleCourant(1) in "*/":
        if SymboleCourant(1) == '*':
            SymboleSuivant(1)
            Facteur()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(nb1 * nb2)
        elif SymboleCourant(1) == '/':
            SymboleSuivant(1)
            Facteur()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(int(nb1 / nb2))

# -----

def Facteur():
    if SymboleCourant(1) == '(':
        SymboleSuivant(1)
        ExprPM()
    if SymboleCourant(1) == ')':
        SymboleSuivant(1)
    else:
        Nombre()

# -----

def Nombre():
    if Chiffre() == True:
        nb = ord(SymboleCourant(1)) - 0x30
        SymboleSuivant(1)
        while Chiffre() == True:
            nb = nb * 10 + ord(SymboleCourant(1)) - 0x30
            SymboleSuivant(1)
        resultat.append(nb)

# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":

```



```

        return True
    else:
        return False

# -----

def popResultat():
    if len(resultat) > 0:
        return resultat.pop()
    else:
        print("Valeur manquante dans l'expression !")
        exit()

# -----

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]

# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----

programme = ("start"
             "512 * (144 + 45412) * 12 - 541 / 2"
             "stop")
resultat = []
posCourante = 0

if Prog() == True:
    print("Le résultat de l'expression est :", popResultat())
else:
    print("Erreur dans le code source à partir de la " + str(posCourante + 1) + "e position !")

```

Cet interpréteur affiche la valeur 279895794 comme résultat de l'exécution de l'expression  $512 * (144 + 45412) * 12 - 541 / 2$ .

## 4.4 Troisième version du langage Z

### 4.4.1 Grammaire

**Ex.** Voici la grammaire donnée déjà dans la section 4.3.1 :

```

<prog>    → start <exprpm> stop
<exprpm>  → <exprfd> | <exprpm> + <exprfd> | <exprpm> - <exprfd>
<exprfd>  → <facteur> | <exprfd> * <facteur> | <exprfd> / <facteur>
<facteur> → ( <exprpm> ) | <nombre>
<nombre>  → <chiffre> | <nombre> <chiffre>
<chiffre> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Voici une version étendue de cette grammaire :

<code>&lt;prog&gt;</code>	<code>→ start &lt;suiteinstr&gt; stop</code>
<code>&lt;suiteinstr&gt;</code>	<code>→ &lt;instr&gt;   &lt;suiteinstr&gt; ; &lt;instr&gt;</code>
<code>&lt;instr&gt;</code>	<code>→ &lt;reg32&gt; = &lt;exprpm&gt;</code>
<code>&lt;exprpm&gt;</code>	<code>→ &lt;exprfd&gt;   &lt;exprpm&gt; + &lt;exprfd&gt;   &lt;exprpm&gt; - &lt;exprfd&gt;</code>
<code>&lt;exprfd&gt;</code>	<code>→ &lt;facteur&gt;   &lt;exprfd&gt; * &lt;facteur&gt;   &lt;exprfd&gt; / &lt;facteur&gt;</code>
<code>&lt;facteur&gt;</code>	<code>→ ( &lt;exprpm&gt; )   &lt;nombre&gt;   &lt;reg32&gt;</code>
<code>&lt;nombre&gt;</code>	<code>→ &lt;chiffre&gt;   &lt;nombre&gt; &lt;chiffre&gt;</code>
<code>&lt;chiffre&gt;</code>	<code>→ 0   1   2   3   4   5   6   7   8   9</code>
<code>&lt;reg32&gt;</code>	<code>→ eax   ebx   ecx   edx   esi   edi</code>

Avec cette nouvelle grammaire :

- L'opérateur d'affectation = permet de transférer dans un registre général de 32 bits le résultat de l'évaluation d'une l'expression. Comme cet opérateur est moins prioritaire que les opérateurs + et -, la règle `<instr> → <reg32> = <exprpm>` fait référence au symbole non-terminal `<exprpm>`. Il n'y a aucune récursivité au niveau de l'opérateur d'affectation, car il ne peut être présent qu'une et une seule fois dans une instruction d'affectation.
- Pour pouvoir avoir plusieurs instructions d'affectation, on trouve de la récursivité au niveau du symbole non-terminal `<suiteinstr>`. Le symbole terminal `;` sert de séparateur entre une instruction d'affectation et la suivante.
- Au niveau du `<facteur>`, on peut avoir maintenant une expression entre parenthèses, un nombre ou un registre général de 32 bits.

## 4.4.2 Interpréteur

Voici l'interpréteur construit à partir de la nouvelle version de la grammaire :

```
def Prog():
    SymboleSuivant(0)
    if SymboleCourant(5) == "start":
        SymboleSuivant(5)
        SuiteInstr()
        if SymboleCourant(4) == "stop":
            SymboleSuivant(4)
            return True
        else:
            return False
    else:
        return False

# -----

def SuiteInstr():
    global numInstr
    Instr()
    while SymboleCourant(1) == ';':
        SymboleSuivant(1)
        numInstr = numInstr + 1
        Instr()

# -----
```

```

def Instr():
    if Reg32() == True:
        regCible = SymboleCourant(3)
        SymboleSuivant(3)
        if SymboleCourant(1) == '=':
            SymboleSuivant(1)
            ExprPM()
            resultatInstrs.append([regCible, popResultat()])

# -----

def ExprPM():
    ExprFD()
    while SymboleCourant(1) in "+-":
        if SymboleCourant(1) == '+':
            SymboleSuivant(1)
            ExprFD()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(nb1 + nb2)
        elif SymboleCourant(1) == '-':
            SymboleSuivant(1)
            ExprFD()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(nb1 - nb2)

# -----

def ExprFD():
    Facteur()
    while SymboleCourant(1) in "*/":
        if SymboleCourant(1) == '*':
            SymboleSuivant(1)
            Facteur()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(nb1 * nb2)
        elif SymboleCourant(1) == '/':
            SymboleSuivant(1)
            Facteur()
            nb2 = popResultat()
            nb1 = popResultat()
            resultat.append(int(nb1 / nb2))

# -----

def Facteur():
    if SymboleCourant(1) == '(':
        SymboleSuivant(1)
        ExprPM()
        if SymboleCourant(1) == ')':
            SymboleSuivant(1)
    elif Reg32() == True:
        i = 0
        while i < len(resultatInstrs) and resultatInstrs[i][0] != SymboleCourant(3):
            i = i + 1
        if i < len(resultatInstrs):
            resultat.append(resultatInstrs[i][1])
            SymboleSuivant(3)
        else:
            print(SymboleCourant(3),
                  "utilisé avant initialisation dans l'instruction", numInstr, "!")
            exit()
    else:
        Nombre()

```

```
# -----

def Nombre():
    if Chiffre() == True:
        nb = ord(SymboleCourant(1)) - 0x30
        SymboleSuivant(1)
        while Chiffre() == True:
            nb = nb * 10 + ord(SymboleCourant(1)) - 0x30
            SymboleSuivant(1)
        resultat.append(nb)

# -----

def Chiffre():
    if SymboleCourant(1) in "0123456789":
        return True
    else:
        return False

# -----

def Reg32():
    if SymboleCourant(3) in ["eax", "ebx", "ecx", "edx", "esi", "edi"]:
        return True
    else:
        return False

# -----

def popResultat():
    if len(resultat) > 0:
        return resultat.pop()
    else:
        print("Valeur manquante dans l'instruction", numInstr, "!")
        exit()

# -----

def SymboleCourant(n):
    return programme[posCourante:posCourante + n]

# -----

def SymboleSuivant(n):
    global posCourante
    posCourante = posCourante + n
    while posCourante < len(programme) and programme[posCourante] == ' ':
        posCourante = posCourante + 1

# -----

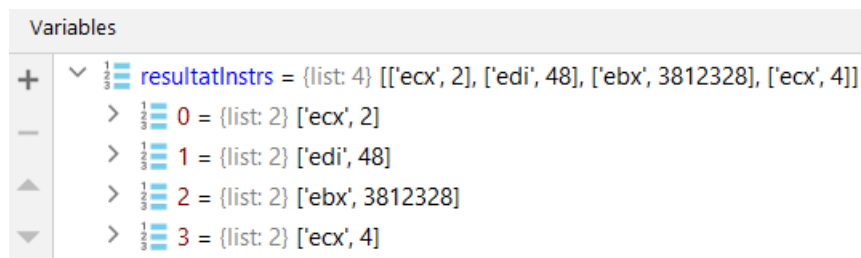
programme = ("start"
             "ecx = 4 / 2;"
             "edi = 45 + 6 / ecx;"
             "ebx = 512 * (144 + ecx) * 51 - edi / 2;"
             "ecx = ecx + ecx"
             "stop")
resultat = []
resultatInstrs = []
numInstr = 1
posCourante = 0

if Prog() == True:
    for element in resultatInstrs:
        print("Le registre", element[0], "stocke la valeur", element[1])
else:
```

```
print("Erreur dans le code source à partir de la " + str(posCourante + 1) +
      "e position dans l'instruction", numInstr, "!")
```

Quelques commentaires :

- La liste *resultatInstrs* stocke le résultat se trouvant dans chaque registre cible après l'exécution des différentes instructions d'affectation qui constituent le programme source. Voici une vue du contenu de la liste *resultatInstrs* dans le débogueur à la fin de l'exécution de l'interpréteur :



- La liste *resultat* est utilisée pour stocker les résultats intermédiaires pendant l'exécution de de l'instruction d'affectation courante.
- La variable *numInstr* stocke le numéro de l'instruction d'affectation en cours d'exécution.

Voici les résultats affiché par l'interpréteur :

<i>Programme source en langage Z</i>	<i>Résultats affichés à l'écran</i>
<pre>start   ecx = 4 / 2;   edi = 45 + 6 / ecx;   ebx = 512 * (144 + ecx) * 51 - edi / 2;   ecx = ecx + ecx stop</pre>	<pre>Le registre ecx stocke la valeur 2 Le registre edi stocke la valeur 48 Le registre ebx stocke la valeur 3812328 Le registre ecx stocke la valeur 4</pre>

# Bibliographie

Aho A. et Ullman J. (1994). *Foundations of Computer Science*. W. H. Freeman.

Swinnen G. (2012). *Apprendre à programmer avec Python 3*. Eyrolles.