

LA CRÉATION D'UN COMPILATEUR POUR LE LANGAGE PYTHON





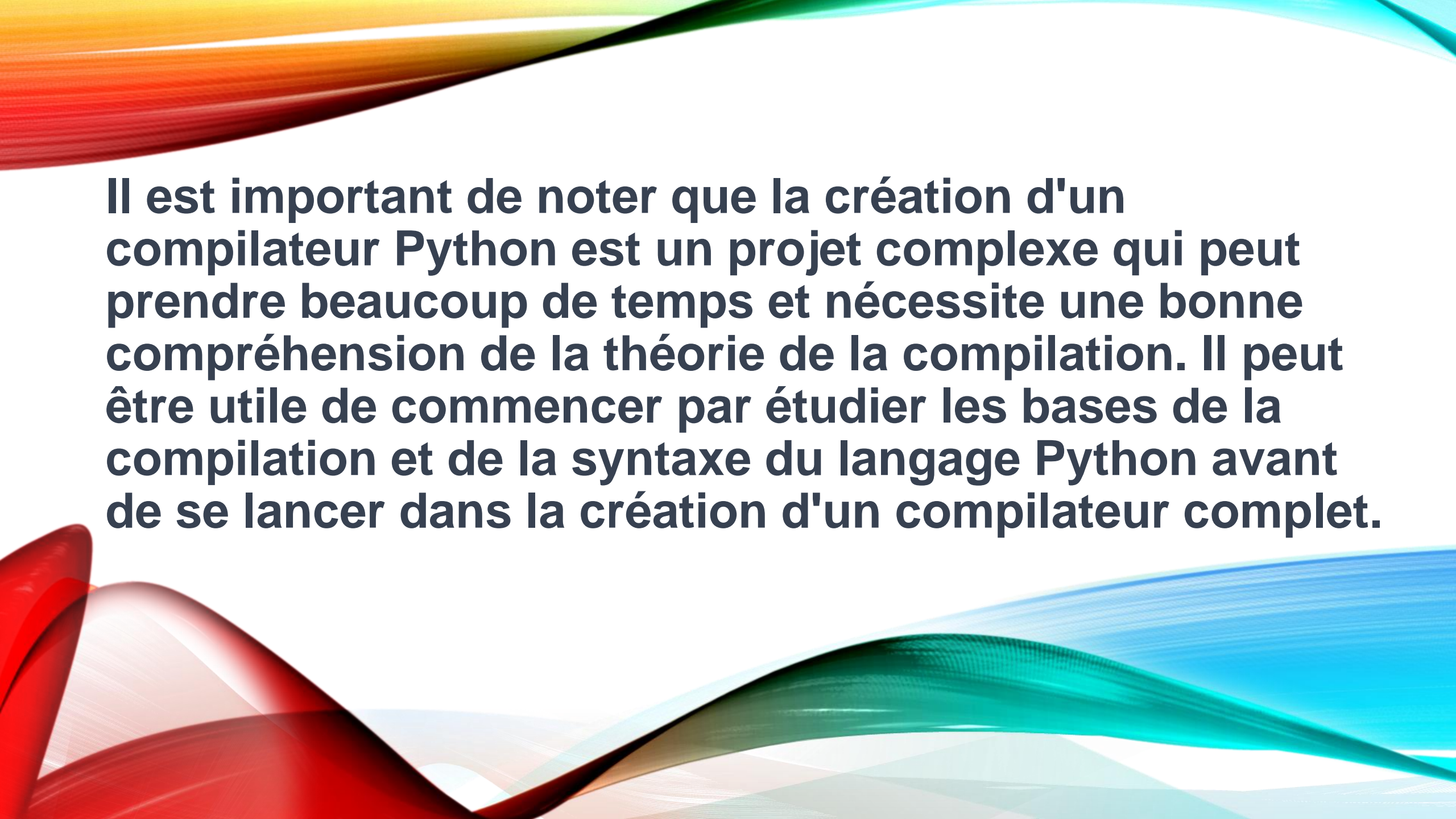
LES ÉTAPES GÉNÉRALES À SUIVRE POUR CRÉER UN COMPILATEUR DE LANGAGE PYTHON :

- **Analyse lexicale** : la première étape consiste à analyser le code source Python pour identifier les différents lexèmes (mots clés, identificateurs, opérateurs, etc.) utilisés dans le code.
- **Analyse syntaxique** : après avoir identifié les lexèmes, il faut utiliser une grammaire formelle pour vérifier si le code respecte les règles syntaxiques du langage Python. C'est à cette étape que l'on peut détecter les erreurs de syntaxe.
- **Analyse sémantique** : une fois que la syntaxe du code a été validée, il est nécessaire de vérifier la sémantique du code. Cela implique de vérifier si les types sont corrects, si les variables sont bien déclarées, etc.



LES ÉTAPES GÉNÉRALES À SUIVRE POUR CRÉER UN COMPILATEUR DE LANGAGE PYTHON :

- **Génération de code intermédiaire** : après avoir validé la syntaxe et la sémantique, il est temps de générer du code intermédiaire qui peut être exécuté sur une machine virtuelle.
- **Optimisation de code** : il est possible d'optimiser le code généré en utilisant différentes techniques pour améliorer les performances.
- **Génération de code final** : une fois que le code intermédiaire a été optimisé, il est possible de générer le code final qui peut être exécuté sur la machine cible.
- **Exécution du code** : le code final peut maintenant être exécuté sur la machine cible.



Il est important de noter que la création d'un compilateur Python est un projet complexe qui peut prendre beaucoup de temps et nécessite une bonne compréhension de la théorie de la compilation. Il peut être utile de commencer par étudier les bases de la compilation et de la syntaxe du langage Python avant de se lancer dans la création d'un compilateur complet.



ANALYSEUR LEXICAL

Supposons que nous voulons créer un analyseur lexical pour un langage qui ne contient que des nombres entiers et des opérateurs arithmétiques de base (+, -, *, /).

1. Tout d'abord, nous allons importer la bibliothèque Python "re" (pour les expressions régulières) qui nous permettra de définir des motifs et pour identifier des motifs dans notre entrée.

En python : `" import re"`

2. Nous allons ensuite définir une expression régulière pour identifier les nombres entiers. Dans notre exemple, nous allons considérer que les nombres sont composés uniquement de chiffres. Notre expression régulière sera donc simplement `"\d+"`.

En python : `" INTEGER = re.compile(r'\d+') "`

3. Ensuite, nous allons définir des expressions régulières pour identifier les opérateurs arithmétiques de base. Dans notre exemple, nous avons besoin de trois expressions régulières, une pour chaque opérateur :


En python : `"PLUS = re.compile(r'\+')
MINUS = re.compile(r'\-')
MULTIPLY = re.compile(r'*')
DIVIDE = re.compile(r'\/')"`

4. Maintenant que nous avons défini nos expressions régulières, nous pouvons créer notre analyseur lexical. Pour ce faire, nous allons écrire une fonction "tokenize" qui prendra une chaîne d'entrée et renverra une liste de "tokens". Un "token" est simplement un objet qui représente une unité lexicale de notre langage, comme un nombre entier ou un opérateur.

```
def tokenize(input_string):
    tokens = []
    position = 0
    while position < len(input_string):
        # essayons de faire correspondre chaque expression régulière à partir de la position actuelle
        match = INTEGER.match(input_string, position)
        if match:
            # si nous avons trouvé un nombre, nous créons un token "INTEGER" avec la valeur correspondante
            tokens.append(('INTEGER', int(match.group(0))))
            position = match.end(0)
            continue
        match = PLUS.match(input_string, position)
        if match:
            # si nous avons trouvé un opérateur "+", nous créons un token "PLUS"
            tokens.append(('PLUS', match.group(0)))
            position = match.end(0)
            continue
        match = MINUS.match(input_string, position)
        if match:
            # si nous avons trouvé un opérateur "-", nous créons un token "MINUS"
            tokens.append(('MINUS', match.group(0)))
            position = match.end(0)
            continue
        match = MULTIPLY.match(input_string, position)
        if match:
            # si nous avons trouvé un opérateur "*", nous créons un token "MULTIPLY"
            tokens.append(('MULTIPLY', match.group(0)))
            position = match.end(0)
            continue
        match = DIVIDE.match(input_string, position)
        if match:
            # si nous avons trouvé un opérateur "/", nous créons un token "DIVIDE"
            tokens.append(('DIVIDE', match.group(0)))
            position = match.end(0)
            continue
        # si nous n'avons trouvé aucun motif, nous émettons une erreur
        raise ValueError('Invalid input at position {}'.format(position))
    return tokens
```



ANALYSEUR SYNTAXIQUE



Nous avons déjà un analyseur lexical qui renvoie une liste de tokens pour une entrée donnée. Nous allons maintenant créer un analyseur syntaxique qui prendra cette liste de tokens et vérifiera si elle correspond à la grammaire de notre langage.

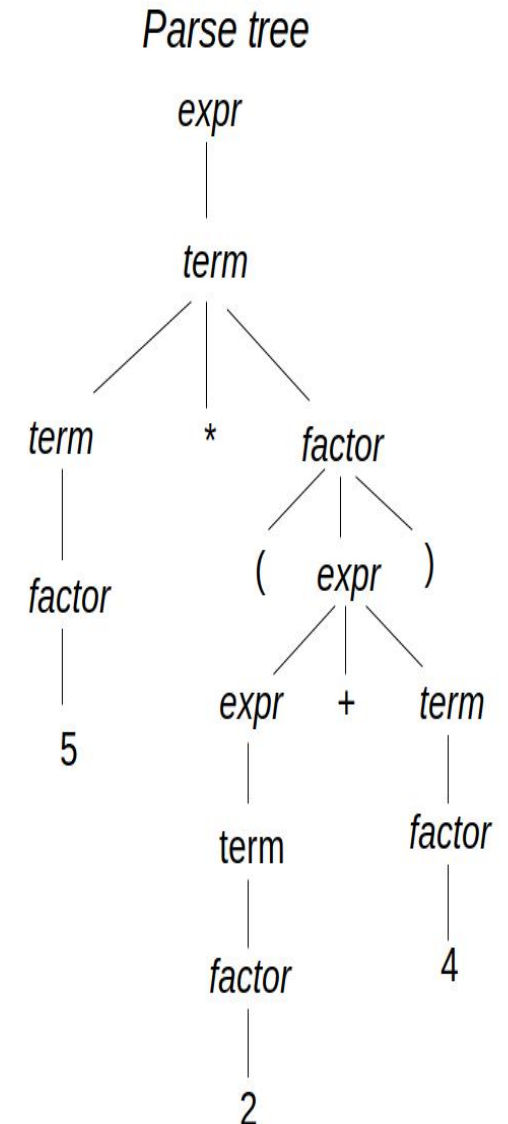
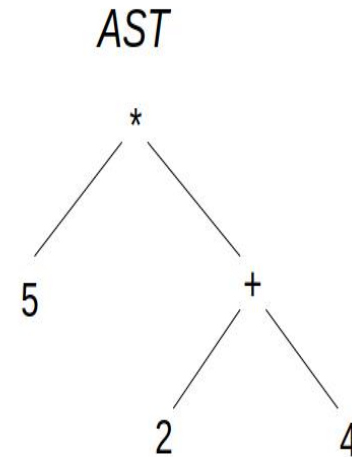
Notre langage ne contient que des expressions arithmétiques simples composées uniquement de nombres entiers et d'opérateurs arithmétiques de base (+, -, *, /).

$$5 * (2 + 4)$$

1. Tout d'abord, nous allons définir une fonction "parse_expression" qui prendra une liste de tokens et renverra l'expression arithmétique correspondante sous forme d'un arbre de syntaxe abstraite (AST).

Un AST est une structure de données qui représente la structure syntaxique d'une expression sous forme d'un arbre.

```
En python : << def parse_expression(tokens):  
# pour le moment, nous ne traitons que  
#les expressions avec un seul terme  
if len(tokens) == 1 and tokens[0][0] == 'INTEGER':  
return ('INTEGER', tokens[0][1])  
raise ValueError('Invalid expression') >>
```





2. Maintenant, nous allons créer une fonction "parse" qui prendra une chaîne d'entrée et renverra l'AST correspondant pour l'expression arithmétique.

En python : << `def parse(input_string):`
 `# tokenize l'entrée`
 `tokens = tokenize(input_string)`
 `# analyse syntaxique`
 `ast = parse_expression(tokens)`
 `# retourne l'AST`
 `return ast` >>

3. Nous pouvons maintenant tester notre analyseur syntaxique avec quelques exemples :

```
En python : << # expression arithmétique valide : 2 + 3
              input_string = '2 + 3'
              ast = parse(input_string)
              print(ast) # ('PLUS', ('INTEGER', 2), ('INTEGER', 3))

              # expression arithmétique invalide : 2 + * 3
              input_string = '2 + * 3'
              try:
                  ast = parse(input_string)
              except ValueError as e:
                  print(str(e)) # Invalid input at position 4 >>
```

Dans cet exemple, notre analyseur syntaxique est très simple, car notre langage ne contient que des expressions arithmétiques simples.

Cependant, pour les langages plus complexes, l'analyse syntaxique peut être beaucoup plus compliquée.

RÉSUMÉ

L'analyseur syntaxique (*parser*, en anglais) est le programme informatique qui réalise cette tâche.

Cette opération suppose une formalisation du texte, qui est vue le plus souvent comme un élément d'un langage formel, défini par un ensemble de règles de syntaxe formant une grammaire formelle .

La structure révélée par l'analyse donne alors précisément la façon dont les règles de syntaxe sont combinées dans le texte.

Cette structure est souvent une hiérarchie de syntagmes, représentable par un arbre syntaxique .