Language Design

Découvrir la conception de **langage de programmation**

Robin Penea

Planning

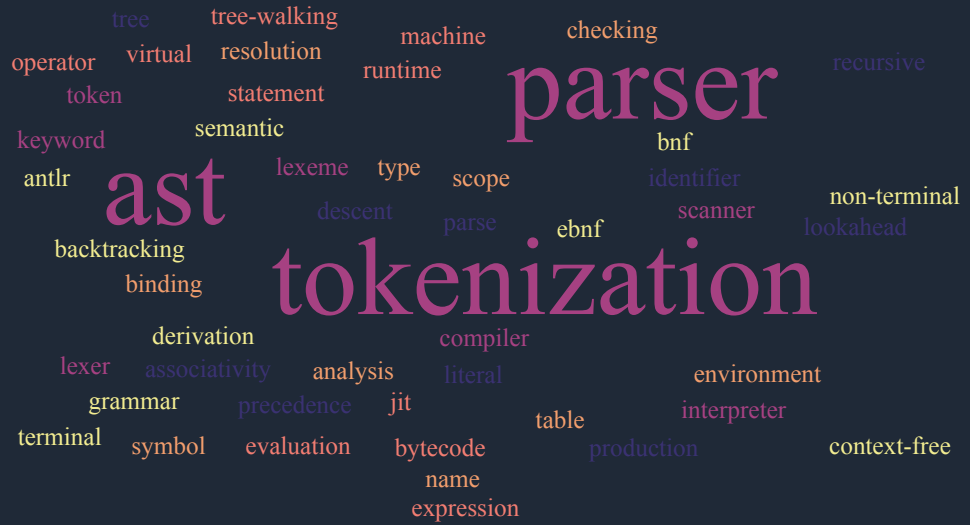
- Cours lundi, mardi, mercredi
- **Démarrage projets mercredi après-midi**
- Suivi projets lundi
- **Rendu projets vendredi semaine prochaine !**

Sommaire

- Fondamentaux PL
- Lexer
- Parser
- Interpréteur

🔥 Démo Toy REPL 🔥

Comment aborder ce cours ?



Beaucoup de défis

Domaine très, très vaste

- Existe depuis littéralement le début de l'informatique (années 50)
- Comparé au web qui est *relativement* jeune (fin années 90)
- Des doctorats sont faits sur l'optimisation des boucles 🤯

Beaucoup de défis

Et pour vous...

- Complètement **étranger** à ce que vous faites d'habitude
- Il faut condenser beaucoup d'information en quelques jours
- On va **tout faire from scratch**. **TOUT**.
  - Nous aurons une seule dépendance, pour les tests
  - Il y a un morceau que nous allons générer, par manque de temps sûrement 😭

Comment aborder ce cours ?

- On réduit le scope (je vais revenir là dessus)
- Je vais **entrelacer** les différentes phases
  - Pas une approche bulldozer (chaque phase est entièrement implémentée)
  - Plutot une approche **agile** (des A/R pour comprendre les interactions entre phases)
- Pratique, pratique, pratique

Utilisation des LLMs

Pour être + fort, pas seulement produire +

- Un LLM peut **très** facilement vous générer du code ici
  - Problème bien borné
  - GitHub regorge de repos de langages de programmation

Si vous voulez progresser

VOUS DEVEZ écrire plusieurs langages, manuellement

Utilisation des LLMs

Pour être + fort, pas seulement produire +

- Nous allons utiliser du code généré
- Je vous indiquerai quand les LLMs deviennent très pertinent
- Par défaut, n’acceptez pas ce que propose l’autocomplete
  - 9/10, il m’a sorti des choses qui **ressemblaient** à ce que je voulais
  - … Mais c’était souvent faux

Dernier point

Que vous apporte ce cours concrètement ?

Ce cours pour vous

- J'ai eu l'idée de faire ce cours à cause de l'impact qu'a eu l'étude du PL design sur mon **propre dev**
- Je sens que je ne **perçois** plus la programmation tout à fait pareil qu'avant
- Très concrètement, il y a (très) peu de chance que vous deviez concevoir des PL dans votre quotidien de dev
- Mais...

Ce cours pour vous

- L'approche de séparation par phase vous servira dans **tous vos devs**
- Réaliser un DSL (**D**omain **S**pecific **L**anguage) ne vous fera pas peur
- Vous allez mieux comprendre **pourquoi** les langages que vous utilisez ont ces forces / faiblesses
  - "Ah oui, c'est une `statement`, pas une `expression`"
  - "Je vois, c'est optimisé à la compilation !"
  - "Ca c'est juste du **sucre syntaxique**"

C'est un  de dev à dev

Et maintenant

On fonce dans le code ! 👩‍💻

Calculatrice PN

Les fondamentaux du **P**rogramming **L**anguage (PL) design

Calculatrice PN

La PN (**P**olish **N**otation) est une syntaxe de programmation **simplifiée**.

Au lieu de :

```
1 + 2
```

On écrit :

```
+ 1 2
```

L'opérateur `+` qui est normalement infixe (se place entre les opérandes)

devient préfixe (se place devant l'opérande).

Pourquoi faire du PN ?

Simplifie notre code à cause de la gestion de la précédence des opérateurs

```
# Quelle opération faut-il faire en 1er ?
1 + 2 * 4

# Si on le fait à la main
1 + (2 * 4)
```

L'opérateur `*` a une **precedence** plus grande que `+` (il est prioritaire).

Avec le PN:

```
+ 1 * 2 4
```

Les étapes sont :

1. `+` de `1` avec
2. Le résultat de `2 * 4`

Pourquoi faire du PN ?

- Pas de gestion de priorité
- Pas de parenthèses à mettre
- Pas d’arbre syntaxique à construire

On peut démarrer sans devoir tout implémenter dès le début

Création du projet `calc-pn`

- IDE de votre choix (PyCharm, VS Code, Neovim, etc.)
- **Python >= 3.10**
- Création d'un `venv`
- 1 fichier `calc.py`

Phases de la calculatrice

Tokenization

- Conversion du `texte` en `token`
- Un `token` est un morceau de texte avec quelques méta données supplémentaires :
  - Type (Nombre, Opérateur, mot clé, etc.)
  - Valeur (lexeme)
  - Ligne dans le code (1, 2, 3, etc.)

⚠️ **A cette étape, il n'y a** aucune compréhension **du code**

Pour le moment, j'omets volontairement toute notion d'interpréteur, compilateur, etc.

Fondamentaux des PLs

- Les PLs vont quasi systématiquement découper leur traitement par phase
- Chaque phase **prépare** le travail pour la phase suivante
- Offre des possibilités incroyables
  - Refacto (remplacement d'une phase sans impacter le reste)
  - Optimisations (une phase d'optimisation est très commune)

Imaginez une pipeline,
chaque phase prend un `input`, fait un traitement,
et produit un `output` pour la phase suivante

Cette approche a reconfiguré mon cerveau

Tokenization

- Définition ensemble de :
  - `Tokens`
  - `TokenType`
  - `tokenize()`
- Test unitaire sur `tokenize()`
  - Installation `pytest`
  - Création `test_calc.py`

Tokenization

```python
# test_calc.py

def test_tokenizer_empty_source_noop():
    source = ""
    assert tokenize(source) == []


def test_tokenizer_simple_addition():
    source = "+ 1 2"
    assert tokenize(source) == [
        Token(TokenType.PLUS, "+"),
        Token(TokenType.NUMBER, "1"),
        Token(TokenType.NUMBER, "2"),
    ]
```

Phases de la calculatrice

Evaluate

- Phase où on va traiter les `tokens` pour les **interpréter**
- Parcours des `tokens` pour obtenir le résultat de l'opération mathématique
- Cette phase attend une liste de `Tokens` et produit un `Number`

Evaluator

- `evaluate()` avec support `+` et `-`
- `calculate()` qui combine `tokenize()` et `evaluate()`
- Tests dans `test_calc.py`

```
def test_calculate_empty_raises_exception():
    source = ""
    with pytest.raises(ValueError):
        calculate(source)


def test_calculate_simple_addition():
    source = "+ 1 2"
    assert calculate(source) == 3
```

TP #01

- Implémenter les opérateurs `mult` et `div`
- Gérer l'erreur de division par 0
  - `ZeroDivisionError`
- Implémenter les tests suivants :

```python
def test_calculate_div_by_zero():
    source = "/ 1 0"
    with pytest.raises(ZeroDivisionError):
        calculate(source)


def test_calculate_sequence_of_operations():
    source = "- + 3 4 1"  # (3 + 4) - 1
    assert calculate(source) == 6
```

Qu'est-ce qu'un interpréteur ?

tokenization + evaluation

(oui c'est tout)

Qu’est-ce qu’un interpréteur ?

- Dans notre PL, nous allons vouloir exprimer des choses plus complexes :
  - Précedence des opérateurs
  - Control flow (`if`, `while`, etc.)
  - Fonctions
- Il y a bien-sur un océan de complexité supplémentaire
- Mais en substance, voilà ce que fait un interpréteur

Toy

Notre langage de programmation interprété

A quoi ressemble Toy ?

```
// Greet a person based on the time of day
fn greet(name, hour) {
    var greeting = "";
    if (hour < 12) {
        greeting = "Good morning";
    } else {
        if (hour < 18) {
            greeting = "Good afternoon";
        } else {
            greeting = "Good evening";
        }
    }

    print(greeting + ", " + name + "!");
}

var theGreeter = greet;  // Oooooh 🤯

theGreeter("Bob", 9);   // Good morning, Bob!
theGreeter("Alice", 15);  // Good afternoon, Alice!
theGreeter("Robin", 21);  // Good evening, Robin!
```

A quoi ressemble Toy ?

- Déclaration de variable avec lexical scoping
- Expressions arithmétiques
- Commentaires
- Control flow (`if` , `while` , `for` , etc)
- First-class functions

A quoi ressemble Toy ?

Sur la syntaxe, je dirais que Toy est banal. Aucun surprise. Un mix simplifié de :

- Kotlin (mot clé `var`)
- JS (manipulation des fonctions)
- Python (mot clé `and`, `or`)

Et c'est exactement ce que l'on veut pour apprendre à faire un PL !

Les idées folles pour vos langages

Pour tenir en 2,5 jours...

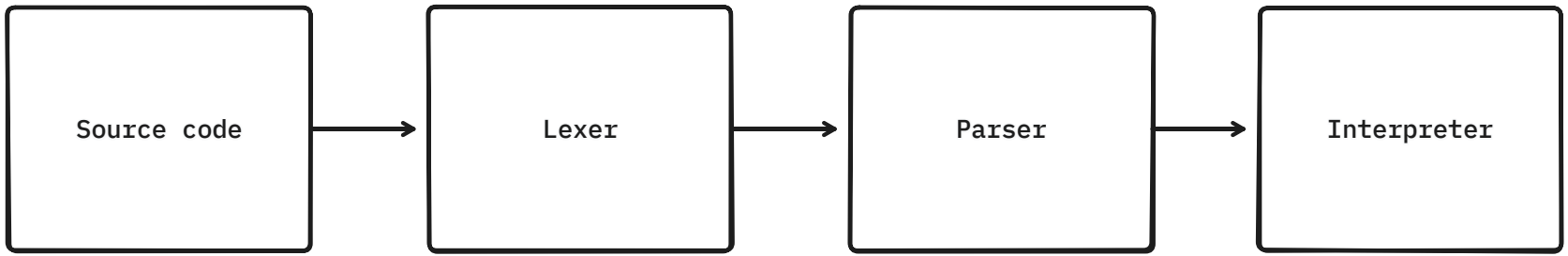
Nous allons devoir prendre des raccourcis

Nous allons devoir prendre des raccourcis

- Pas de gestions de string
- Pas d'opérateurs logique (`and` , `or`)
- Pas de commentaires
- Pas d'implémentation manuelle du lexer + parser de fonction

Le projet final contient toutes les étapes !

Phases de notre interpréteur




```
var age = 21;
if (age > 18) {
  print "Majeur";
} else {
  print "Oops";
}
```

```
[var] [age] [=]
[21] [;] [if] [(]
[age] [>] [18]...
```

```
VarDec("age", 21)
Binary(
  cond=Comp("age", 18),
  then=Print("majeur"),
  else=Print("Oops")
)
```

```
"Majeur"
```

Module 1
Fondations - Lexer

- Tokenization : source → tokens
- Scanner caractère par caractère
- Types de tokens (NUMBER, opérateurs, EOF)
- Architecture du lexer

Lexer

- Plusieurs termes pour la même notion :
  - Lexer (**LEX**ical analyz**ER**)
  - Scanner
  - Tokenizer
- Objectif : découper le code en `token`
- Notre `Lexer` suit la meme logique que pour la calculatrice, avec un peu plus de complexité

Lexer - Fondamentaux

- Démarrage projet `toy`
- Définition des tokens de base (`+`, `-`, `NUMBER`, etc)
- Implémentation classe `Lexer`
  - `tokenize()`
  - `scan()`
  - `peek()`
  - ...
- Mise en place de tests unitaires

TP #02 - Étendre le lexer

Objectif: Implémenter la gestion des whitespaces, newlines et opérateurs `*` et `/`

À faire:

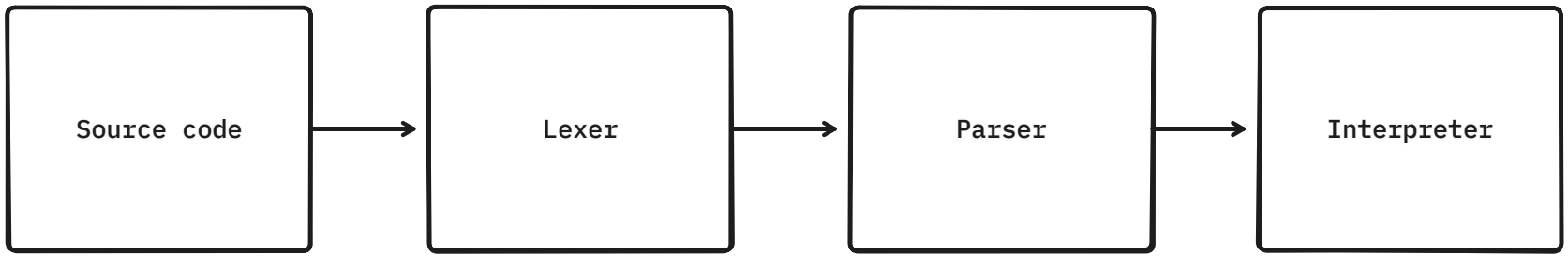
- Dans `scan()`, ignorer (`pass`) des caractères suivants `" " | "\t" | "\n"`
- Gérer `*` et `/` dans le lexer
- Le test suivant doit passer

```
1   def test_lexer_tokenize_ignored_chars():
2       source = "2\t * 3\n / 4"
3       lexer = Lexer(source)
4       tokens = lexer.tokenize()
5       assert tokens == [
6           Token(TokenType.NUMBER, "2", 1),
7           Token(TokenType.STAR, "*", 1),
8           Token(TokenType.NUMBER, "3", 1),
9           Token(TokenType.SLASH, "/", 2),
10          Token(TokenType.NUMBER, "4", 2),
```

Module 2
Parsing & AST

- **A**bstract **S**yntax **T**ree (AST)
- Recursive descent parsing
- Précédence des opérateurs
- Hiérarchie expression → term → factor → primary

Phases de notre interpréteur




```
var age = 21;
if (age > 18) {
  print "Majeur";
} else {
  print "Oops";
}
```

```
[var] [age] [=]
[21] [;] [if] [(]
[age] [>] [18]...
```

```
VarDec("age", 21)
Binary(
  cond=Comp("age", 18),
  then=Print("majeur"),
  else=Print("Oops")
)
```

```
"Majeur"
```

Pourquoi un Parser ?

- Un PL exprime des constructions plus **complexes** qu'une calculatrice
- Nous devons prémâcher le travail pour la phase suivante : **l'interpréteur**
- Dans quel **ordre** faut-il exécuter le code suivant ?

```
1   var age = 15;
2   if (age + 5 >= minAge()) {
3       print "OK";
4   }
```

- Il faut d'abord **résoudre** la condition pour savoir si on affiche `"OK"` :
  1. Faire `age + 5` ⇒ `20`
  2. Appeler la fonction `minAge()` pour récupérer le résultat ⇒ `18`
  3. Faire la comparaison `20 >= 18` ⇒ `true`

Pourquoi un Parser ?

- Le rôle du Parser est de :
  - Prendre la liste de `tokens` en `input`
  - Produire une structure que l'interpréteur peut parcourir dans le bonne ordre en `output`
- Cette structure s'appelle...

AST

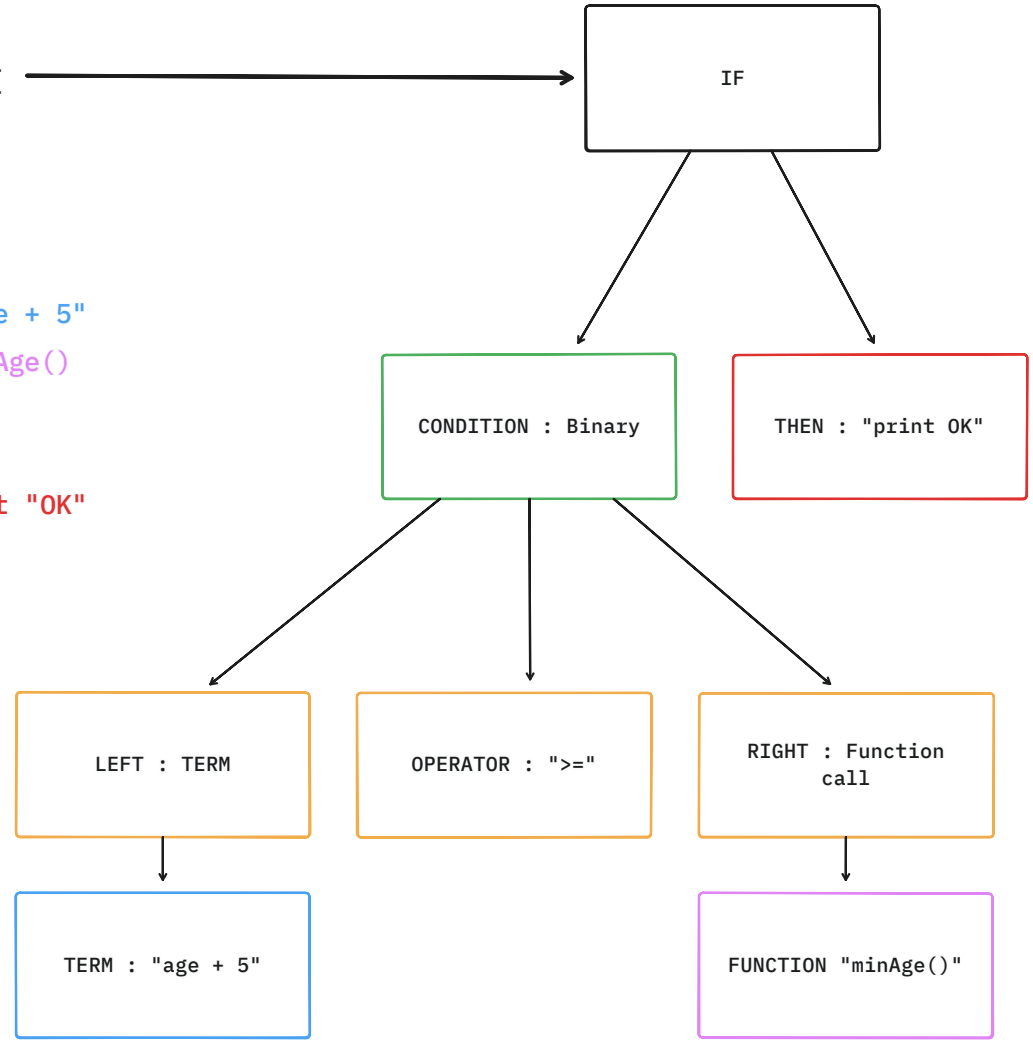
Abstract Syntax Tree

La pièce maîtresse

(Je reviendrai dessus, mais cela qui m'a reconfiguré le cerveau)

```
var age = 15;
if (age + 5 >= minAge()) {
    print "OK";
}
```

1. Résoudre opération "age + 5"
2. Résoudre return de minAge()
3. Comparer 1. et 2.
4. Finir la CONDITION
5. Exécuter le THEN: print "OK"



AST

- Cet arbre sera parcouru par l'interpréteur en partant des **feuilles** jusqu'à la **racine**
- Pour simplifier, chaque ligne de code avec un `;` est une racine
- Un programme est donc une liste d'instructions qui sont des racines ⇒ `list[ASTNode]`
- Pour générer cette arbre, nous avons 2 questions critiques :
  - Comment décider des priorités dans le code ?
  - Comment générer l'arbre dans le bon ordre ?

AST

- Il existe beaucoup d'algorithmes pour générer un AST :
  - Recursive descent parser
  - Pratt parser
  - Shunting-yard algorithm
  - Et tellement d'autres 😲
- Tous ont leurs avantages / inconvénients en fonction de la situation

AST

- On peut mixer les algos :
  - **Go** : Recursive descent pour les statements, Pratt pour les expressions
  - **Rust** : Recursive descent pour la structure globale, Pratt pour les opérateurs
  - **TypeScript** : Recursive descent pour les statements, precedence climbing pour expressions
- Nous allons partir sur le Recursive descent parser

Recursive descent

Principe

- Chaque règle de grammaire = une méthode
- Les méthodes s'appellent entre elles de manière **récursive**
- La hiérarchie des appels détermine la précédence

Définitions

- **Grammaire** : Les règles de syntaxe du langage (comment écrire du code valide)
- **Précédence** : Quel opérateur calculer en premier (`*` avant `+`, etc.)

Recursive descent

Hiérarchie (du moins au plus prioritaire)

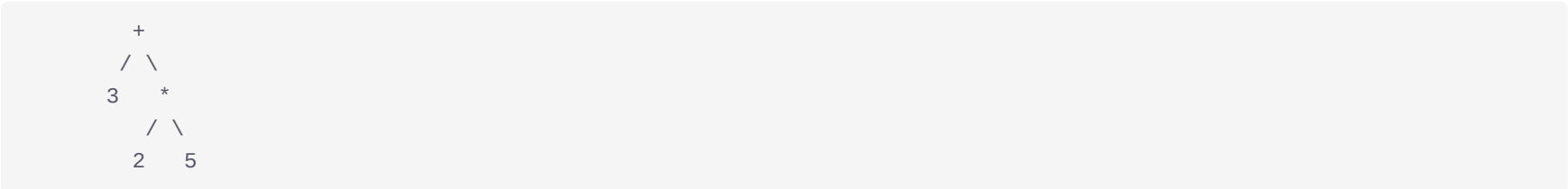
```
expression() → term() → factor() → primary()
 (entrée)      (+ -)     (* /)     (nombres)

Exemple : 3 + 2 * 5
1. expression() appelle term()
2. term() appelle factor()
3. factor() appelle primary() → lit 3
4. Remonte à term(), voit + (pas géré ici, remonte)
5. expression() voit +, rappelle term()
6. Nouveau term() → factor() → primary() lit 2
7. factor() voit * (géré ici !), rappelle factor() → lit 5
8. ✨ Résultat : 3 + (2 * 5) — * est traité avant + !
```

Schéma d'appel

```
1   expression()
2       │
3       ├─> term()
4       │     └─> factor()
5       │           └─> primary() → lit '3'
6       │
7       ├─ voit '+' ─┐
8       │            │
9       └─> term()   │ (après le +)
10            │
11            ├─> factor() ──────┐ voit '*' !
12            │     └─> primary() → lit '2'
13            │                  │
14            └─> factor()  <────┘ (après le *)
15                  └─> primary() → lit '5'
```

Recursive descent

Résultat AST



Points clés

- On descend toujours jusqu'à `primary()` pour lire un nombre
- `factor()` gère `*` et `/` (haute priorité)
- `term()` gère `+` et `-` (basse priorité)
- La structure force `2 * 5` à être groupé avant d'être additionné à `3`

Live coding | Parser `term` et `primary`

- Définition structure `ast_nodes.py`
  - `ASTNode` et `Expression`
  - `Literal` et `Binary`
- Implémentation recursive descent parser
- Parsing de `term` (addition/soustraction)
- Parsing de `primary` (littéraux et parenthèses)
- `expression()` → `term()` → `primary()`

Live coding | Parser `term` et `primary`

Dans `test_toy.py`

```
1   def test_parse_term():
2       source = "3 + 2"
3       lexer = Lexer(source)
4
5       tokens = lexer.tokenize()
6       parser = Parser(tokens)
7       ast = parser.parse()
8
9       assert ast == [
10          Binary(
11              Literal(3.0),
12              Token(TokenType.PLUS, "+", 1),
13              Literal(2.0),
14          ),
15      ]
```

Précédence des opérateurs

- Problème: `2 + 3 * 4` doit être `2 + (3 * 4)`
- Notre structure avec le recursive descent le résout naturellement
  - L'ordre d'appel des fonctions définie la priorité
  - Hiérarchie: `term()` (priorité basse) → `factor()` (priorité haute) → `primary()`
  - Concrètement, on insère `factor()` dans la chaine d'appel
- Associativité gauche de ces opérateurs
  - Les opérateurs de meme priorité sont regroupés vers la gauche
  - `1 + 2 + 3` devient `(1 + 2) + 3`

Live coding | Parser Factor

- Parsing de `factor` (multiplication/division)
- Nouvelle hiérarchie: `term()` → `factor()` → `primary()`
- Refactoring du helper `binary_left()`

Live coding | Parser Factor

Dans `test_toy.py`

```
1   def test_parse_factor():
2       source = "3 + 2 * 4"
3       lexer = Lexer(source)
4
5       tokens = lexer.tokenize()
6       parser = Parser(tokens)
7       ast = parser.parse()
8
9       assert ast == [
10          Binary(
11              Literal(3.0),
12              Token(TokenType.PLUS, "+", 1),
13              Binary(
14                  Literal(2.0),
15                  Token(TokenType.STAR, "*", 1),
16                  Literal(4.0),
17              ),
18          ),
19      ]
```

Comment gérer les `(` ?

Pas si dur que ça 😏

Comment gérer les `(` ?

Problème `(3 + 2) * 4`

- Avec notre implémentation actuelle, nous faisons `3 + (2 * 4)`
  - (En fait notre implémentation crash, on ne gère pas les parenthèses)
- Les parenthèses nous indiques que nous devons parser une expression
  - `(expression) * 4`
  - Une parenthèses n'est ni un `term`, ni un `factor`
  - ... C'est donc un `primary` !

Comment gérer les `(` ?

Pas si dur que ça 😏

Comment gérer les (?

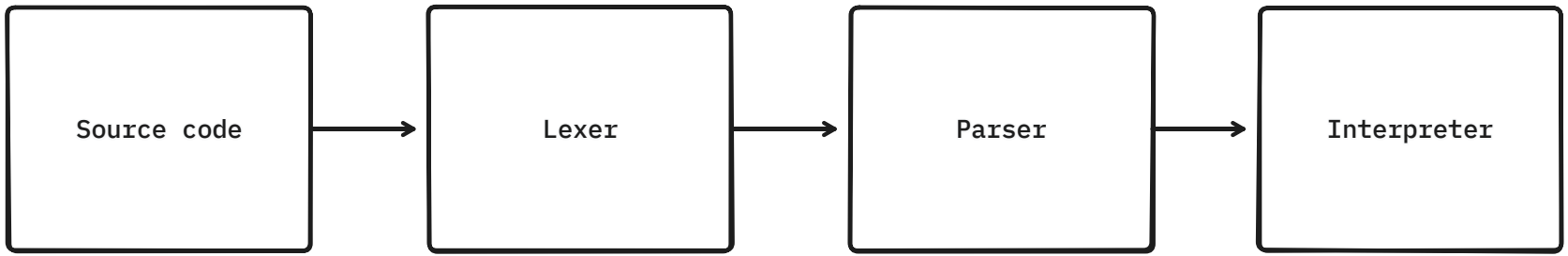
Dans `test_toy.py`

```
def test_parse_factor_with_parenthesis():
    source = "(3 + 2) * 4"
    lexer = Lexer(source)

    tokens = lexer.tokenize()
    parser = Parser(tokens)
    ast = parser.parse()

    assert ast == [
        Binary(
            Binary(
                Literal(3.0),
                Token(TokenType.PLUS, "+", 1),
                Literal(2.0),
            ),
            Token(TokenType.STAR, "*", 1),
            Literal(4.0),
        )
    ]
```

Module 3
Interpréteur de base

- Tree-walking interpreter
- Évaluation récursive
- Opérateurs arithmétiques et comparaison

Phases de notre interpréteur




```
var age = 21;
if (age > 18) {
  print "Majeur";
} else {
  print "Oops";
}
```

```
[var] [age] [=]
[21] [;] [if] [(]
[age] [>] [18]...
```

```
VarDec("age", 21)
Binary(
  cond=Comp("age", 18),
  then=Print("majeur"),
  else=Print("Oops")
)
```

```
"Majeur"
```

Interpreter

- L'interpréteur est la dernière pièce du puzzle
- Prend l'AST généré par le Parser en `input`
- Évalue le résultat en output

Pour le moment, il ne gère que les expressions

Pour rappel :

- Un expression produit un résultat (`3 + 2` ⇒ `5` , `3 > 2` ⇒ `true`)
- Un statement produit un effet (`print "Coucou";`)
  - 👆 Nous reviendrons sur cette notion rapidement

Live coding | Interpreter - Arithmétique

- Mise en place squelette `interpreter.py`
- Méthode `evaluate()` avec pattern matching
- Évaluation de `Literal` et `Binary`
- Test des 4 opérations (+, -, *, /)

Live coding | Interpreter - Arithmétique

Dans `test_toy.py`

```python
def test_evaluate_factor():
    source = "3 + 2 * 4"  # 3 + (2 * 4)

    lexer = Lexer(source)
    tokens = lexer.tokenize()
    parser = Parser(tokens)
    ast = parser.parse()
    interpreter = Interpreter()

    result = None
    for statement in ast:
        result = interpreter.evaluate(statement)
    assert result == 11.0
```

Objectif Ajouter les opérateurs `>`, `>=`, `<`, `<=`, `==`

Lexer

- Implémenter `Lexer.match()` pour lookahead d'un caractère (meme logique que dans `Parser`)
- Exemple, quand on match `>` :
  - Si `self.match(TokenType.EQUAL)`, on retourne `TokenType.GREATER_EQUAL`
  - Sinon, on retourne `TokenType.GREATER`

```
1       def match(self, expected: str) -> bool:
2           if self.is_at_end():
3               return False
4           if self.source[self.current] != expected:
5               return False
6           self.current += 1
7           return True
```

TP #03 - Compléter les comparaisons

Parser + Interpreteur

- Nous avons 2 nouvelles règles : `parse_equality()` et `parse_comparison()`
- Ordre d'appel :

`parse_expression()` ⇒ `parse_equality()` ⇒ `parse_comparison()` ⇒ `parse_term()`

- Toujours utiliser notre helper `binary_left()`

TP #03 - Compléter les comparaisons

Dans `test_toy.py`

```python
# Parser
def test_parse_comparison_equality():
    source = "3 > 2 == 4"  # (3 > 2) == 4
    lexer = Lexer(source)
    tokens = lexer.tokenize()
    parser = Parser(tokens)
    ast = parser.parse()
    assert ast == [
        Binary(
            Binary(
                Literal(3.0),
                Token(TokenType.GREATER, ">", 1),
                Literal(2.0),
            ),
            Token(TokenType.EQUAL_EQUAL, "==", 1),
            Literal(4.0),
        )
    ]
```

TP #03 - Compléter les comparaisons

```python
# Interpreter
def test_evaluate_comparison():
    source = "3 + 2 > 4"
    lexer = Lexer(source)
    tokens = lexer.tokenize()
    parser = Parser(tokens)
    ast = parser.parse()
    interpreter = Interpreter()

    result = None
    for statement in ast:
        result = interpreter.evaluate(statement)

    assert result == True
```

```python
# Interpreter
def test_evaluate_equality():
    source = "3 + 2 == 4"
    lexer = Lexer(source)
    tokens = lexer.tokenize()
    parser = Parser(tokens)

    ast = parser.parse()
    interpreter = Interpreter()

    result = None
    for statement in ast:
        result = interpreter.evaluate(statement)

    assert result == False
```

Opérateurs unaires - et !

- Node `Unary` dans l'AST
- Associativité droite : `--3` ⇒ `-(-3)` ⇒ `3`
- On démarre ensemble
  - Ajout des tokens `MINUS`, `BANG` dans le lexer
  - Refacto des tests ensemble

Live coding | Opérateur unaire début

```python
def tokenize(source: str) -> list[Token]:
    """Tokenize source code."""
    lexer = Lexer(source)
    return lexer.tokenize()


def parse(source: str) -> list:
    """Parse source code into AST."""
    tokens = tokenize(source)
    parser = Parser(tokens)
    return parser.parse()


def evaluate(source: str):
    """Evaluate source code and return result."""
    ast = parse(source)
    interpreter = Interpreter()
    result = None
    for statement in ast:
        result = interpreter.evaluate(statement)
    return result
```

```python
# Interpreter tests
@pytest.mark.parametrize(
    "source,expected",
    [
        ("3 + 2 * 4", 11.0),  # Precedence: 3 + (2 * 4)
        ("3 + 2 > 4", True),  # Comparison: 5 > 4
        ("3 + 2 == 5", True),  # Equality: 5 == 5
        ("3 + 2 == 4", False),  # Equality: 5 == 4
    ],
)
def test_evaluate_expressions(source, expected):
    assert evaluate(source) == expected
```

TP #04 - Terminer les unaires

- Ajouter `Unary` dans `ast_nodes.py`
- `parse_factor()` ⇒ `parse_unary()` ⇒ `parse_primary()`
- Ajouter évaluation de `Unary` dans `Interpreter.evaluate()`

```python
# Interpreter tests
@pytest.mark.parametrize(
    "source,expected",
    [
        ("3 + 2 * 4", 11.0),  # Precedence: 3 + (2 * 4)
        ("3 + 2 > 4", True),  # Comparison: 5 > 4
        ("3 + 2 == 5", True),  # Equality: 5 == 5
        ("3 + 2 == 4", False),  # Equality: 5 == 4
        ("-2 + 3", 1),  # Unary: 1
    ],
)
def test_evaluate_expressions(source, expected):
    assert evaluate(source) == expected
```

Module 4
Variables & État

- **Changement conceptuel majeur!**
- Statements vs Expressions
- Environment (symbol table)
- Mutation de variables

Statements vs Expression

Point critique du PL design

- Jusque là nous avons manipulé des expressions
- Pour pouvoir ajouter la déclaration de variables, nous devons introduire la notion de statement

Statement vs Expression

Une expression est va toujours produire une valeur

```
1   // Arithmétique
2   3 + 2           // => 5
3
4   // Comparaison
5   3 > 2           // => true
6   5 == 5          // => true
7
8   // Logique
9   true and false  // => false
10  true or false   // => true
11
12  // Variables
13  x               // => valeur de x
14  a + 1           // => valeur de a + 1
15
16  // Assignment (aussi une expression!)
17  a = 5           // => 5 (et modifie a)
18  b = a = 10      // => 10 (assignment chaining)
19
20  // Appels de fonction
```

Statement vs Expression

Un statement va modifier l'état du programme

```
1   // Déclaration
2   var x = 5;
3   var name = "Bob";
4
5    // Expression statement (expression + point-virgule)
6   a = 2;
7   print 42;
8
9   // Control flow
10  if (x > 5) { print x; }
11  while (i < 5) { i = i + 1; }
12  for (var i = 0; i < 5; i = i + 1) { print i; }
13
14  // Déclarer une fonction
15  fn greet(name) {
16      print "Hello, " + name;
17  }
```

Statement vs Expression

- C'est un point critique, car il va déterminer la structure de notre PL
- Ce qui est considéré comme une expression peut etre **assigné à une variable**
- Au delà des mots-clés (`var` , `fn` , etc), l'ergonomie du langage est aussi influencée par ces choix

```
1   var result = 3 + 2;      // 3 + 2 est une expression, var result = ... est un statement
2   print x > 5;             // x > 5 est une expression, print ... est un statement
3   var a = b = 10;          // b = 10 est une expression qui retourne 10
```

Statement vs Expression

- Pour Toy, nous allons simplifier les choses
- Tout ce qui finit par un `;` est un statement, donc...
  - `3 + 2` est une expression
  - `3 + 2;` est un statement
- Nous allons devoir wrapper notre `Expression` dans un `ExpressionStatement`

Live coding | Déclaration variable

- Statement `var name = expression;`
- Ajouter ce qui manque au `Lexer` : `identifier()` avec le lien `tokens.KEYWORDS`
- AST nodes: `Statement`, `VarStatement`
- Nouveau point d'entrée dans `Parser` : `declaration()`
  - `parse()` ⇒ `parse_declaration()` ⇒ `parse_var_statement()` ⇒ `parse_expression()`

Dans `test_toy.py`

```
1   def test_parse_var_declaration():
2       ast = parse("var a = 1;")
3       assert ast == [
4           VarStatement(
5               Token(TokenType.IDENTIFIER, "a", 1),
6               Literal(1.0),
```

Live coding | ExpressionStatement

- Encapsulation de `Expression` dans `ExpressionStatement`
- AST Node : `ExpressionStatement`
- `Parser.parse_declaration()` a un fallback sur `parse_statement()`
  - `parse_declaration()` ⇒ `parse_statement` ⇒ `parse_expression_statement()` ⇒ `parse_expression()`

Live coding | ExpressionStatement

Avant

```
1   def test_parse_comparison_operators():
2       ast = parse("3 > 2")
3       assert ast == [
4           Binary(
5               Literal(3.0),
6               Token(TokenType.GREATER, ">", 1),
7               Literal(2.0),
8           )
9       ]
```

Après

```
1   def test_parse_comparison_operators():
2       ast = parse("3 > 2;")
3       assert ast == [
4           ExpressionStatement(
5               Binary(
6                   Literal(3.0),
7                   Token(TokenType.GREATER, ">", 1),
8                   Literal(2.0),
9               )
10          )
11      ]
```

`Environment`

Stockage de l'interpréteur

`Environment`

- Comment l'interpreteur doit traiter le node `VarStatement` ?
- Concrètement, il a besoin d'un espace de stockage RAM pour :
  - Stocker les variables
  - Récupérer les valeurs des variables
  - Stocker les fonctions
  - Exécuter le code associé à chaque fonction

`Environment`

- L'approche la plus évidente, un dictionnaire associant :
  - `name` ⇒ `value`
- Nous aurons besoin de plus dans le futur (lexical scoping, closure, etc.)
- Classe `Environment` qui wrappe notre dictionnaire
  - Méthodes: `define()`

Terminologie

- Dans un interpréteur, on dira plutôt un Environment
- Dans un compilateur, on dira plutôt une Symbol table

Live coding | Définition Environment

- class `Environment` avec méthode `define()`
- Déclaration d'un environnement global dans `Interpreter`
- Dans `Interpreter`, ajouter `execute()`, qui complète la structure
  - `interpret()` ⇒ `execute()` ⇒ `evaluate()`
  - Traitement de `ExpressionStatement`
  - Traitement de `VarStatement`

Assignement

Continuité naturelle

On veut pouvoir écrire :

```
1   var a = 1;      // Dans Environment => { "a": 1 }
2   a = 2;          // Dans Environment => { "a": 2 }
```

- Intuitivement, on dirait que c'est un statement...
- ... Il est tout à fait **juste** de dire que l'assignement modifie l'état du programme

Assignement

Continuité naturelle

Mais, si veut pouvoir écrire :

```
1   // Assignment en tant qu'expression permet le chaining
2   var a = b = c = 10;  // Tous valent 10
3
4   // Utiliser l'assignment dans une condition
5   // On assigne ET on teste en une seule ligne
6   var line;
7   while ((line = readLine()) != null) {
8       print line;
9   }
10
11  // Assignment dans un argument de fonction
12   print(x = 5);  // Affiche 5 et assigne x
```

Assignement doit produire une valeur

Assignement

Continuité naturelle

- Notre `VariableAssignment` sera donc une `Expression`
- Assignment: `VariableAssignment` node
- `Environment.assign()` vs `define()`
- Assignment chaining (`a = b = 3`)

Live coding | Parser `assignment`

- On démarre ensemble, et je vous laisse terminer l'implémentation
- Pour `a = 1;`, nous devons :
  - "Retrouver" qui est `a`
  - Gérer l'expression d'égalité qui est une assignation

Live coding | Parser `assignment`

- Ajout de l'ASTNode `Variable`, `VariableAssignment`
- Dans `Parser`
  - Parsing de `Variable` dans `parse_primary()`
  - Ajout de `parse_assignment()`
  - `parse_expression()` ⇒ `parse_assignment()` ⇒ `parse_equality()`

Live coding | Parser `assignment`

```python
def parse_assignment(self) -> Expression:
    expr = self.parse_equality()

    # On match si c'est =
    # Si oui, on stock equals = self.previous()
        # On parse la value (avec parse_assignement()), récursivité
        # Si l'expr est une instance de variable,
            # on retourne VariableAssignment avec le token et l'expr

    return expr
```

Dans `Parser`

- Implémenter parsing de `assignment()` (identifier `=` expression)

```
def test_parse_var_assignment():
    ast = parse("var a = 1; a = 2;")
    assert ast == [
        VarStatement(
            Token(TokenType.IDENTIFIER, "a", 1),
            Literal(1.0),
        ),
        ExpressionStatement(
            VariableAssignment(
                Token(TokenType.IDENTIFIER, "a", 1),
                Literal(2.0),
            )
        ),
    ]
```

Dans `Environment`

- Ajouter une méthode pour assigner une valeur a une variable :

```python
def define(self, name: str, value: Any) -> None:
```

- Ajouter une méthode pour récupérer la valeur d'une variable :

```python
def get(self, name: str) -> Any:
```

⚠️ Dans les 2 cas, raise `RuntimeError` si la variable n'existe pas

(on veut aider les devs, pas leur laisser la possibilité de se tirer une balle dans le pied)

Dans `Interpreter`

- Ajouter le traitement dans `evaluate()`
  - `Variable` ⇒ Renvoyer la valeur pour la nom donné
  - `VariableAssignment` ⇒ Modifier la valeur pour la nom donné ET **renvoyer la nouvelle valeur**

```python
@pytest.mark.parametrize(
    "source,expected",
    [
        ("3 + 2 * 4;", 11.0),  # Precedence: 3 + (2 * 4)
        ("3 + 2 > 4;", True),  # Comparison: 5 > 4
        ("3 + 2 == 5;", True),  # Equality: 5 == 5
        ("3 + 2 == 4;", False),  # Equality: 5 == 4
        ("-2 + 3;", 1),  # Unary: 1
        ("var a = 1; a + 1;", 2),  # VariableAssignment: 2
    ],
)
```

Print statement

Plutôt ...

```
1   var a = 10;
2   print a;  // Affiche 10, print est un mot clé
```

Ou... ?

```
1   var a = 10;
2   print(a);   // Affiche 10, print est une fonction
```

c'est une décision de PL design majeure !

Print statement

- Chaque langage a son approche
  - Python : `print` mot-clé en Python 2, puis `print()` fonction en Python 3 (le bon choix technique, l'enfer de la communauté)
  - Go : `fmt.Println()` fonction
  - Rust : `println!()` macro
  - JavaScript : `console.log()` fonction
- Limites du mot-clé ?
  - Impossible de **mocker** un mot-clé. `print` fait partie de l'ADN du langage 😬
  - Une fonction est plus **cohérent**, tout appel de fonction a la même syntaxe

En nous on fait quoi ?

Print statement pour Toy

- `print` en keyword nous permet d'avoir un vrai REPL plus rapidement
- … Sinon, on doit avoir toute la mécanique de fonction **avant** de pouvoir l'utiliser 🥵
- En plus, nous pourrons utiliser **l'Official Toy REPL** !

TP #06 - Print

- `Lexer` : rien à faire ☺️
- Dans `ast_nodes.py` ⇒ `PrintStatement` qui a une `expression`
- Dans `parser.py` ⇒ `parse_print_statement()`
  - `parse_statement()` qui l'appel si `match(TokenType.PRINT)`
  - Fallback sur la fonction existante `parse_expression_statement()`

`parse_print_statement()`

```
1   def parse_print_statement(self) -> Statement:
2       expr = self.parse_expression()
3       self.consume(TokenType.SEMICOLON, "Expect ';' after expression.")
4       return PrintStatement(expr)
```

C'est tellement élégant 🥲

TP #06 - Print

Dans `test_toy.py`

```
1   def test_parse_print_statement():
2       ast = parse("print 1 + 2;")
3       assert ast == [
4           PrintStatement(
5               Binary(
6                   Literal(1.0),
7                   Token(TokenType.PLUS, "+", 1),
8                   Literal(2.0),
9               )
10          )
11      ]
```

TP #06 - Print

Dans `Interpreter`

- Implémenter `PrintStatement` dans `execute()` :
  - Evaluer l'expression
  - print(expression)
- Vérifier dans `main.py` que tout est bien cablé, vous deviez pouvoir faire :

```
PYTHONPATH=src python src/toy/main.py
```

TP #06 - Print

Pour utiliser l'**Official Toy REPL** :

Installer les dépendances

```
pip install textual
```

Copier les `tools/` **à côté** do `src/`, pour avoir:

```
my-app
  ├── src/
  └── tools/
```

Lancer le REPL

```
python tools/repl/repl.py
```

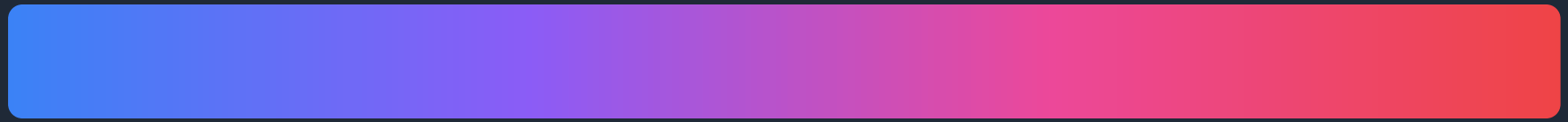
ℹ️ Le REPL est **future compatible**, il supportera vos prochaines versions de Toy sans modifier le code.

Interpréteur vs Compilateur

Un spectre, pas une dichotomie

Le spectre Interpréteur ↔ Compilateur

Ce n’est **pas** une opposition binaire, mais un **gradient** avec de nombreuses nuances



Interpréteur pur **Bytecode + VM** **JIT** **Compilateur pur**

Les deux extrêmes du spectre

Interpréteur pur

- Exécution **directe** du code source ou de l'AST
- Pas de phase de compilation séparée
- Évaluation statement par statement

Exemples:

- Notre **Toy** (tree-walking interpreter)
- Python (CPython)
- Ruby (MRI)
- Shells (bash, zsh)

Compilateur pur

- Traduction **complète** en code machine natif
- Phase de compilation **avant** l'exécution
- Fichier exécutable indépendant (binaire)

Exemples:

- C, C++
- Rust
- Go
- Fortran

Bytecode + VM (Machine Virtuelle)

- **Compilation** vers un format intermédiaire : le **bytecode**
- Le bytecode = instructions simplifiées, indépendantes de la plateforme
- Une **VM** (Virtual Machine) interprète le bytecode

Exemples: Python (`.pyc`), Java (JVM), C# (.NET), Lua

```
# Code Python                  # Bytecode approximatif
x = 10 + 20          →         LOAD_CONST 10
                               LOAD_CONST 20
                               BINARY_ADD
                               STORE_NAME x
```

JIT (Just-In-Time Compilation)

- Compilation **dynamique** pendant l'exécution
- Optimisations basées sur le profiling runtime (hot paths)
- Combine flexibilité de l'interprétation + performance de la compilation

Exemples: JavaScript (V8, SpiderMonkey), Java HotSpot, PyPy, LuaJIT

Avantages et inconvénients

Critère	Interpréteur	Compilateur
Temps de démarrage	✅ Rapide (pas de compilation)	❌ Lent (compilation requise)
Performance runtime	❌ Plus lent (overhead interprétation)	✅ Très rapide (code machine natif)
Portabilité	✅ Code source = portabilité	❌ Binaire par plateforme
Debugging	✅ Erreurs ligne par ligne, REPL	⚠️ Plus complexe (optimisations)
Feedback développeur	✅ Hot-reload instantané	❌ Cycle compile-run-debug
Distribution	⚠️ Besoin de l'interpréteur installé	✅ Binaire standalone
Optimisations	❌ Limitées (pas de vue globale)	✅ Agressives (inlining, DCE, etc.)

Pourquoi la compilation est à la mode ?

Contexte moderne

1. **Performance critique** : Applications modernes (web, cloud, ML, gaming) ont besoin de vitesse
2. **Matériel abordable** : La compilation n'est plus un goulot (machines rapides, CI/CD parallèle)
3. **Outils modernes** : Compilateurs rapides (Rust, Go compilent vite), hot-reload même compilé
4. **Sécurité & Robustesse** : Langages avec vérification statique de types (Rust, TypeScript)

Pourquoi la compilation est à la mode ?

Mais les interpréteurs restent essentiels pour :

- Prototypage rapide et expérimentation
- Scripts (DevOps, automation, data science)
- REPLs éducatifs (comme notre Toy !)
- Langages embarqués (Lua dans jeux vidéo, config files)

Et si on voulait le compiler ?

Bonne nouvelle: On peut garder le même front-end (Lexer, Parser, AST) !

Seul le **back-end** change :

- Au lieu d'un `Interpreter` qui traverse l'AST
- On crée un `Compiler` qui **génère du bytecode**
- Puis une **VM** qui exécute le bytecode

Gain de performance: x3 **instantanément** (moins d'overhead, instructions optimisées)

Module 5
Control Flow

- Conditionnelles (`if` / `else`)
- Blocks et scoping
- While loops
- For loops via desugaring

Control Flow

Diriger l'exécution du programme

- Jusqu'ici, notre code s'exécute ligne par ligne
- Le **control flow** permet de :
  - Exécuter conditionnellement : `if` / `else`
  - Répéter : `while`, `for`
  - Sortir d'une fonction : `return`

If/Else - Sémantique

```
var age = 15;
if (age >= 18) {
    print "Adulte";
} else {
    print "Mineur";
}
// Affiche : Mineur
```

- Évaluer la **condition** (expression)
- Si truthy : exécuter `then_branch`
  - `true` ⇒ La valeur booléenne `true`
  - `truthy` ⇒ `!0`, `"non empty string"`, `!null`, `!undefined`, etc.
- Sinon : exécuter `else_branch` (optionnel)

Live coding | If/Else

Dans `ast_nodes.py`

- Ajouter `IfStatement`

```
@dataclass
class IfStatement:
    condition: Expression
    then_branch: Statement
    else_branch: Statement | None
```

- Ajouter `BlockStatement(statements)` pour les blocs `{ }`

```
@dataclass
class BlockStatement(Statement):
    statements: list[Statement]
```

Live coding | If/Else

Dans `lexer.py`

- Ajouter tokens `LBRACE`, `RBRACE` pour `{` et `}`

Dans `parser.py`

- `parse_if_statement()`
  - `parse_statement()` ⇒ `parse_if_statement()`
  - Parser condition entre `(` et `)`
  - Parser `then_branch` (statement)
  - Si `else` présent, parser `else_branch`
- `parse_block()` pour `{ statements }`

Live coding | If/Else

Test dans `test_toy.py`

```
def test_parse_if_statement():
    ast = parse("""if (1 == 2) {
                        print 3;
                    } else {
                        print 4;
                    }""")

    assert ast == [
        IfStatement(
            condition=Binary(Literal(1.0), Token(EQUAL_EQUAL, "==", 1), Literal(2.0)),
            then_branch=BlockStatement([PrintStatement(Literal(3.0))]),
            else_branch=BlockStatement([PrintStatement(Literal(4.0))]),
        )
    ]
```

Live coding | Interpreter If/Else

Dans `interpreter.py`

- Ajouter `execute()` pour `IfStatement` :

```
if self.is_truthy(self.evaluate(stmt.condition)):
    self.execute(stmt.then_branch)
elif stmt.else_branch is not None:
    self.execute(stmt.else_branch)
```

- Ajouter `execute()` pour `BlockStatement`
- Ajouter méthode helper `interpret()` pour les tests

Live coding | Interpreter If/Else

Dans `test_toy.py`

```
@pytest.mark.parametrize("source,expected_output", [
    ("if (1 < 2) { print 42; }", "42.0\n"),
    ("if (1 > 2) { print 42; }", ""),
    ("if (1 > 2) { print 1; } else { print 2; }", "2.0\n"),
])
def test_if_statement(capsys, source, expected_output):
    interpret(source)
    captured = capsys.readouterr()
    assert captured.out == expected_output
```

Block Scoping

Variables locales

- Les blocks `{ }` créent un nouveau **scope**
- Cela s'applique partout : `if`, `while`, `for`, ou block standalone
- Variables déclarées dans un block ne sont visibles que dans ce block

```
1   var x = 10;
2   if (true) {
3       var y = 20;  // Scope du if, y disparait quand on sort du block
4   }
5
6   {
7       // La variable local x fait du shadowing à la variable globale x
8       var x = 30;  // Scope du block standalone.
9       print x;     // 30
10  }
11  print x;         // 10
```

Même mécanisme pour tous les `{ }` !

Live coding | Block Scoping

Dans `environment.py`

- Ajouter `parent: Environment | None` au constructeur
- Modifier `get()` : chercher dans le parent si non trouvé localement

```python
def get(self, name: str):
    if name in self.values:
        return self.values[name]
    if self.parent is not None:
        return self.parent.get(name)
    raise RuntimeError(f"Undefined variable '{name}'.")
```

Live coding | Block Scoping

- Modifier `assign()` : chercher dans le parent si non trouvé localement

```python
def assign(self, name: str, value: Any) -> None:
    # On cherche dans l'environnement locale
    if name in self.values:
        self.values[name] = value
        return

    # Puis dans celui du parent (appel récursif => On peut remonter une longue chaine !)
    if self.enclosing:
        self.enclosing.assign(name, value)
        return

    raise RuntimeError(f"Undefined variable '{name}'.")
```

Live coding | Block Scoping

Dans `interpreter.py`

- `execute_block()` crée un nouvel environnement :

```
1   def execute_block(self, statements, environment):
2       previous = self.environment
3       self.environment = environment
4       try:
5           for stmt in statements:
6               self.execute(stmt)
7       finally:
8           # Le finally est important,
9           # S'il y a une exception, on est sûr de restaurer l'environnement initial
10          self.environment = previous
```

Live coding | Block Scoping

Dans `test_toy.py`

```
1   def test_block_environment_scope():
2       source = """
3       var a = 1;
4       var b = 2;
5       {
6           var a = 2;
7           var c = 3;
8           b = 10;
9       }
10      """
11      interpreter = interpret(source)
12
13      # The block scope doesn't affect the global a variable
14      assert interpreter.environment.get("a") == 1.0
15
16      # The b variable is properly resolved in the block scope from the outer scope
17      assert interpreter.environment.get("b") == 10.0
18
19      # The c variable is not defined in the global scope
```

Boucles

Itération dans Toy

- Maintenant qu'on a les conditionnelles, il nous faut des **boucles**
- Une seule vraie boucle : `while`
- Tout le reste ? Du **syntactic sugar** (nous reviendrons rapidement dessus)

While - Sémantique

```
1   var i = 0;
2   while (i < 5) {
3       print i;
4       i = i + 1;
5   }
6   // Affiche : 0, 1, 2, 3, 4
```

- Tant que la condition est truthy, exécuter le body
- Le body s'exécute dans l'environnement courant

En fait, vous avez déjà tout pour le faire !

TP #07 - while

Dans `ast_nodes.py`

```python
@dataclass
class WhileStatement:
    condition: Expression
    body: Statement
```

Dans `parser.py`

- Implémenter `parse_while_statement()`
- `parse_statement()` ⇒ `parse_while_statement()`
- Même logique de parsing que le `if` (consommer `LPAREN`, parser la condition, consommer `RPAREN`, etc.)

TP #07 - while

Dans `interpreter.py`

- Ajouter le cas `WhileStatement` dans `execute()`
  - Boucle : tant que `self.evaluate(condition)`
  - Exécuter `body`

TP #07 - while

Dans `test_toy.py`

```
def test_while_statement(capsys):
    source = """
    var i = 0;
    while (i < 5) {
        print i;
        i = i + 1;
    }
    """
    interpret(source)
    captured = capsys.readouterr()
    assert captured.out == "0.0\n1.0\n2.0\n3.0\n4.0\n"
```

Et les boucles for ?

Syntactic Sugar

Du sucre syntaxique

- **Syntactic sugar** : nouvelle syntaxe, même sémantique
- On transforme le code au parsing vers des constructions existantes
- L'interpréteur n'a rien de nouveau à gérer
- C'est ce qu'on appelle le **desugaring**

Pourquoi ?

Pourquoi le desugaring ?

- Réutilisation : pas de nouveau code dans l'interpréteur
- Simplicité : le runtime reste minimal
- Cohérence : même comportement qu'une boucle while manuelle
- Design de langage : séparer syntaxe surface vs sémantique core

C'est utilisé partout !

Desugaring - Exemples réels

Python : list comprehensions

```python
# Sugar
result = [x * 2 for x in range(5)]

# Desugar
result = []
for x in range(5):
    result.append(x * 2)
```

JavaScript : async/await

```javascript
// Sugar
async function fetchData() {
    const data = await fetch(url);
    return data;
}

// Desugar (simplifié)
function fetchData() {
    return fetch(url).then(data => {
        return data;
    });
}
```

Desugaring - Exemples réels

Kotlin : for-in loop

```kotlin
// Sugar
for (item in list) {
    println(item)
}

// Desugar
val iterator = list.iterator()
while (iterator.hasNext()) {
    val item = iterator.next()
    println(item)
}
```

Rust : if let

```rust
// Sugar
if let Some(value) = option {
    println!("{}", value);
}

// Desugar
match option {
    Some(value) => println!("{}", value),
    None => {}
}
```

For - Desugaring

Ce code dans Toy :

```
for (var i = 0; i < 5; i = i + 1) {
    print i;
}
```

Est transformé en :

```
{
    var i = 0;
    while (i < 5) {
        print i;
        i = i + 1;
    }
}
```

Le `{ }` externe crée un scope pour l'initializer

P.S. Cette approche n'est pas suffisante pour correctement gérer les `break` et `continue`. On va s'en contenter pour le moment.

For - AST généré

Ce code:

```
for (var i = 0; i < 5; i = i + 1) { print i; }
```

Génère cet AST:

```
BlockStatement([
    VarStatement(Token("i"), Literal(0)),
    WhileStatement(
        Binary(Variable("i"), Token("<"), Literal(5)),
        BlockStatement([
            BlockStatement([PrintStatement(Variable("i"))]),
            ExpressionStatement(
                VariableAssignment(Token("i"),
                    Binary(Variable("i"), Token("+"), Literal(1)))
            )
        ])
    )
])
```

Live coding | For desugaring

Dans `parser.py`

- Implémenter `parse_for_statement()`
  - Parser `(` puis 3 parties : `init ; cond ; incr`
  - Parser `)` et le `body`
  - Construire l'AST désugaré :

```
return BlockStatement([
    initializer,
    WhileStatement(
        condition or Literal(True),
        BlockStatement([body, increment])
    )
])
```

Aucun changement dans l'interpréteur !

Live coding | For desugaring

Test dans `test_toy.py`

```
def test_parse_for_statement():
    source = "for (var i = 0; i < 5; i = i + 1) { print i; }"
    ast = parse(source)

    assert ast == [
        BlockStatement([
            VarStatement(Token(..., "i", ...), Literal(0.0)),
            WhileStatement(
                Binary(Variable(...), Token(LESS, "<", ...), Literal(5.0)),
                BlockStatement([
                    BlockStatement([PrintStatement(Variable(...))]),
                    ExpressionStatement(VariableAssignment(...))
                ])
            )
        ])
    ]
```

TP #08 - For desugaring - TP

Objectif: Compléter les cas avec parties optionnelles

À faire:

- Gérer `init` optionnel : `for (; i < 5; i++) { }`
- Gérer `condition` optionnelle : `for (var i = 0; ; i++) { }` → condition devient `Literal(True)`
- Gérer `increment` optionnel : `for (var i = 0; i < 5; ) { }`

Test de scoping à vérifier:

```
var i = 123;
for (var i = 0; i < 1; i = i + 1) {
    print i;  // 0
}
print i;  // 123
```

Le `i` du for ne leak pas en dehors grâce au `{ }` externe

Module 6
Functions

- Fonctions comme valeurs (first-class)
- Closures & environnements
- Appel de fonction
- Return & control flow

Les fonctions

Le cœur d'un vrai langage

- Jusqu'ici, notre code est linéaire
- Les fonctions permettent :
  - **Réutilisation** de code
  - **Abstraction** de logique
  - **Modularité**
- En Toy, les fonctions sont des first-class citizens

First-class citizens ?

Les fonctions sont des valeurs comme les autres

First-class functions

```
1   fn add(a, b) {
2       return a + b;
3   }
4
5   var operation = add;  // Stockée dans une variable
6   operation(3, 5);      // Appelée via la variable
7
8   fn apply(f, x, y) {   // Passée en argument
9       return f(x, y);
10  }
11
12  apply(add, 10, 20);   // → 30
```

Les fonctions sont des valeurs comme les nombres ou les strings

Function Declaration

Définir des fonctions

Function Declaration - Grammaire

AST Node

```python
@dataclass
class FunctionDeclarationStatement:
    name: Token
    parameters: list[Token]  # Liste d'identifiants
    body: list[Statement]    # Corps de la fonction
```

Live coding | Function Declaration

Dans `lexer.py`

- Ajouter token `COMMA` pour séparer les paramètres

Dans `ast_nodes.py`

- Ajouter `FunctionDeclarationStatement(name, parameters, body)`

```python
@dataclass
class FunctionDeclarationStatement(Statement):
    name: Token
    parameters: list[Token]
    # We could use a BlockStatement here, but we will manage
    # the function scope with a custom class
    body: list[Statement]
```

Live coding | Function Declaration

Dans `parser.py`

- `parse_function_declaration()`
  - Parser nom de la fonction
  - Parser `(` et liste de paramètres (séparés par `,`)
  - Parser `)` et le bloc body `{ }`

Dans `interpreter.py`

- Créer classe `ToyFunction(declaration, closure)`
- Exécuter `FunctionDeclarationStatement` → stocker `ToyFunction` dans l'environnement

Function Declaration - ToyFunction

Dans l'interpréteur, on crée un objet `ToyFunction` :

```
@dataclass
class ToyFunction(ToyCallable):
    declaration: FunctionDeclarationStatement
    # Each function has its own environment,
    # where it "closes over" (capture) its surround environment
    # By doing this, we can access variables declared outside the function
    closure: Environment
```

Quand on exécute `FunctionDeclarationStatement` :

```
func = ToyFunction(stmt, self.environment)
self.environment.define(stmt.name.lexeme, func)
```

La fonction capture son environnement → closure !

Live coding | Function Declaration

Test dans `test_toy.py`

```
def test_parse_function():
    source = "fn add(a, b) { return a + b; }"
    ast = parse(source)

    assert ast == [
        FunctionDeclarationStatement(
            name=Token(TokenType.IDENTIFIER, "add", 1),
            parameters=[
                Token(TokenType.IDENTIFIER, "a", 1),
                Token(TokenType.IDENTIFIER, "b", 1),
            ],
            body=[
                ReturnStatement(
                    Token(TokenType.RETURN, "return", 1),
                    Binary(
                        Variable(Token(TokenType.IDENTIFIER, "a", 1)),
                        Token(TokenType.PLUS, "+", 1),
                        Variable(Token(TokenType.IDENTIFIER, "b", 1)),
                    ),
                )
            ],
```

Function Call

Appeler les fonctions

Function Call - Grammaire

AST Node

```python
@dataclass
class FunctionCall:
    callee: Expression        # Ce qu'on appelle
    arguments: list[Expression]
```

Exemple:

```python
add(3, 5)
```

⇒ `FunctionCall(callee=Variable("add"), arguments=[Literal(3), Literal(5)])`

Function Call - Exécution

5 étapes :

1. **Évaluer** le callee → doit être un `ToyFunction`
2. **Évaluer** les arguments (de gauche à droite)
3. **Créer** un nouvel environnement avec la closure comme parent
4. **Lier** les paramètres aux valeurs des arguments
5. **Exécuter** le corps de la fonction

```
1   def call(self, interpreter, arguments):
2       # Créer environnement local
3       env = Environment(parent=self.closure)
4
5       # Lier params aux args
6       for param, arg in zip(self.declaration.parameters, arguments):
7           env.define(param.lexeme, arg)
8
```

Closures en action

```
var x = 5;

fn makeAdder() {
    fn add(a) {
        return a + x;  // x vient de l'environnement parent
    }
    return add;
}

var adder = makeAdder();
adder(3);  // → 8
```

Quand `add` est déclarée, elle capture l'environnement qui contient `x`

C'est ça une closure : fonction + son environnement capturé

Live coding | Function Call

Dans `ast_nodes.py`

- Ajouter `FunctionCall(callee, arguments)`

Dans `parser.py`

- `parse_call()` dans la chaîne de précédence (après `primary`)
  - Si token `LPAREN` après primary → c'est un appel
  - Parser liste d'arguments séparés par `,`
  - Retourner `FunctionCall(callee, args)`

Dans `interpreter.py`

- Évaluer `FunctionCall` :
  - Évaluer le callee (doit être un `ToyFunction`)
  - Évaluer chaque argument

Live coding | Function Call

Dans `ToyFunction.call()`

```python
def call(self, interpreter, arguments):
    # Créer nouvel environnement avec closure comme parent
    env = Environment(parent=self.closure)

    # Lier paramètres aux arguments
    for param, arg in zip(self.declaration.parameters, arguments):
        env.define(param.lexeme, arg)

    # Exécuter le body dans ce nouvel environnement
    interpreter.execute_block(self.declaration.body, env)
```

Live coding | Function Call

Test dans `test_toy.py`

```
def test_function_call(capsys):
    source = """
    var x = 5;
    fn add(a) {
        return a + x;
    }
    var b = add(3);
    print b;
    """
    interpret(source)
    captured = capsys.readouterr()
    assert captured.out == "8.0\n"
```

Return

Sortir d'une fonction avec une valeur

Return - Le problème

```
1   fn max(a, b) {
2       if (a > b) {
3           return a;  // Comment "sauter" hors de la fonction ici ?
4       }
5       return b;
6   }
```

Le `return` doit :

- Arrêter l'exécution du corps de la fonction
- Remonter la valeur au point d'appel
- Même s'il est dans un `if`, un `while`, etc.

C'est du **non-local control flow**

Return - Solution via Exception

On utilise une exception Python !

```python
class ReturnValue(Exception):
    def __init__(self, value):
        self.value = value
```

- Quand on exécute `return expr;` → `raise ReturnValue(valeur)`
- Dans `ToyFunction.call()`, on catch l'exception

```python
def call(self, interpreter, arguments):
    env = Environment(parent=self.closure)
    # ... lier params ...

    try:
        interpreter.execute_block(self.declaration.body, env)
    except ReturnValue as ret:
        return ret.value  # On retourne la valeur

    return None  # Pas de return explicite → None
```

Return - Grammaire

```
statement → "return" expression? ";"
```

AST Node

```python
@dataclass
class ReturnStatement:
    keyword: Token
    value: Expression | None  # Peut être None
```

Exemple:

```
return a + b;   // Avec valeur
return;         // Sans valeur → None
```

Live coding | Return

Dans `ast_nodes.py`

- Ajouter `ReturnStatement(keyword, value)`
  - `value` peut être `None` (return sans valeur)

Dans `interpreter.py`

- Créer exception `ReturnValue(value)` pour le control flow
- Exécuter `ReturnStatement` :
  - Évaluer la `value` (si présente)
  - `raise ReturnValue(valeur)`

Live coding | Return

- Modifier `ToyFunction.call()` :

```
try:
    interpreter.execute_block(self.declaration.body, env)
except ReturnValue as ret:
    return ret.value
return None  # Pas de return → None
```

Live coding | Return

Test dans `test_toy.py`

```python
def test_function_with_return():
    source = """
    fn add(a, b) {
        return a + b;
    }
    """
    interpret(source)
    # La fonction est déclarée sans erreur

def test_function_call_with_return(capsys):
    source = """
    fn add(a, b) { return a + b; }
    var result = add(3, 5);
    print result;
    """
    interpret(source)
    captured = capsys.readouterr()
    assert captured.out == "8.0\n"
```

Module 7
ANTLR

- Parser generators
- Grammaires déclaratives
- Visitor pattern
- Parser manuel vs généré

ANTLR

ANother Tool for Language Recognition

- Jusqu'ici, on a écrit notre parser **à la main**
  - Recursive descent
  - Précédence des opérateurs
  - ~500 lignes de code
- **ANTLR** est un parser generator
  - On écrit une **grammaire**
  - ANTLR génère le lexer + parser
  - On écrit juste un **visitor** pour construire l'AST

Pourquoi utiliser un parser generator ?

Parser manuel vs ANTLR

Parser manuel	ANTLR
Contrôle total	Grammaire déclarative
~500 lignes de code	~150 lignes de grammaire
Debugging facile	Debugging via l'outil (ouch)
Pédagogique ++	Production-ready ++
Flexibilité max	Conventions à suivre

Dans l'industrie : ANTLR, Bison, LALR, PEG parsers, etc.

Pour apprendre : parser manuel d'abord, puis outils

Parser manuel vs ANTLR

Langage	Parser	Détails
Java	Mixte	`javac` : custom / Outils: ANTLR
Python	Généré	PEG généré (depuis 3.9)
Go	Custom	Recursive descent
SQL	Généré	Yacc/Bison (selon SGBD)
Rust	Custom	Recursive descent
C/C++	Mixte	GCC: Bison / Clang: custom
JavaScript	Custom	V8, SpiderMonkey
TypeScript	Custom	Parser propriétaire

Parser manuel vs ANTLR

Quand utiliser un parser generator ?

- La grammaire évolue fréquemment
- Les performances ne sont pas critiques
- **Le front-end (lexer + parser) évolue en même temps que le back-end (compilateur)**

📣 **Le dernier point est la raison pour laquelle je vous montre la génération !**

Grammaire Toy.g4

On définit un fichier de grammaire `Toy.g4` :

```
grammar Toy;

program: statement* EOF;

statement
    : varDeclaration
    | functionDeclaration
    | ifStatement
    | whileStatement
    | forStatement
    | printStatement
    | returnStatement
    | expressionStatement
    | block
    ;

expression
    : assignment
    ;
```

Pipeline ANTLR

Pipeline ANTLR

On lance une pipeline qui va **générer le parser** :

```
./scripts/generate_parser.py  # J'ai déjà préparé le script
```

Cela va :

- Lancer ANTLR pour générer le lexer + parser dans un langage cible (dans notre cas, Python)
- Le code généré se trouve dans `src/toy/generated/`

Le problème étant, **comment brancher avec nos types dans `ast_nodes.py` ?**

Visitor Pattern

Un design pattern 😳 !

Visitor Pattern

Problème :

- D'un côté, on a les classes AST de ANTLR
- De l'autre coté, on a nos classes AST dans `ast_nodes.py`

Solution : Le pattern Visitor

- Sépare l'**algorithme** de la **structure de données**
- Pour chaque type de nœud, on définit une méthode `visit*()`
- Le visitor **traverse** l'arbre et appelle la bonne méthode

Dans notre cas : Parse Tree (ANTLR) → Visitor → AST (nos classes)

ANTLR génère le squelette, on implémente les `visit*()` !

Visitor Pattern

Pipeline complète :

```
Source Code
    ↓
ANTLR Lexer → Tokens
    ↓
ANTLR Parser → Parse Tree
    ↓
Visitor → AST (notre format)
    ↓
Interpreter (inchangé!)
```

L'interpréteur reste identique, on remplace juste le parsing

Visitor Pattern

ANTLR génère un parse tree. On doit le transformer en AST :

```
class ASTBuilder(ToyVisitor):
    def visitProgram(self, ctx):
        statements = []
        for stmt in ctx.statement():
            statements.append(self.visit(stmt))
        return statements

    def visitVarDeclaration(self, ctx):
        name = Token(TokenType.IDENTIFIER, ctx.IDENTIFIER().getText())
        value = self.visit(ctx.expression())
        return VarStatement(name, value)

    # ... un visit*() par règle de grammaire
```

ANTLR dans Toy

- Dans notre cas, nous devons écrire 3 fichiers :
  - `Toy.g4` : la grammaire pour la génération
  - `toy/generated/parser.py` : L'équivalent de notre `Parser` qui fait le lien avec ANTLR (et fait l'appel a `visit()`
  - `toy/generated/visitor.py` : La classe `ASTBuilder` qui implémente le pattern visitor et qui est *longue* (~400 lignes)

Quitte à faire du généré...

ANTLR dans Toy

- J'ai utilisé un Coding Agent pour générer ces 2 classes justement 😎
- Voici un morceau de mon prompt (dispo dans son intégralité dans `project-language-toy/README.md`):

Pour vous projets

- En fonction de vos projets, utiliser ANTLR peut devenir nécessaire pour **pouvoir travailler en parallèle**
- Ecrivez manuellement :
  - Des exemples de codes avec le maximum de syntaxe
  - Le fichier `tokens.py`
  - Le fichier `ast_nodes.py`
  - Le squelette de votre `parser.py`
- Ensuite, utilisez le LLM pour :
  - Générer la grammaire ANTLR
  - **PUIS** Générer le parser (qui fait appel au visitor)

Pour vos projets

- Cette approche permet à l'équipe travaillant sur l'interpreteur / back-end de commencer a travailler tout de suite
- Pendant ce temps, l'équipe du front-end pourra écrire le lexer + parser manuellement
- **Tant que la grammaire est instable, c'est un gain de temps**

Module 8
Polishing

- Strings
- Opérateurs logiques (`and`, `or`)
- Commentaires
- Built-in functions (`clock()`, `number()`)

Finitions

Rendre Toy utilisable

- Notre langage est fonctionnel
- Mais il manque des **features essentielles** pour être pratique
- On va ajouter rapidement :
  - **Strings** : manipulation de texte
  - `and` / `or` : logique booléenne avec court-circuit
  - **Commentaires** : documenter le code
  - **Built-ins** : fonctions natives (`clock()`, conversions)

Strings

Ajout:

- Token `STRING` dans le lexer : `"hello world"`
- AST : `Literal(value: str)`
- Opérateur `+` pour concaténation

Exemple:

```
var name = "Alice";
var greeting = "Hello, " + name + "!";
print greeting;  // Hello, Alice!
```

Opérateurs logiques

`and` et `or` avec court-circuit

```
var x = false and print("never executed");
var y = true or print("never executed");
```

- `and` : si gauche est `false`, ne pas évaluer la droite
- `or` : si gauche est `true`, ne pas évaluer la droite

Implémentation:

```
def evaluate_logical(self, expr):
    left = self.evaluate(expr.left)

    if expr.operator.type == TokenType.OR:
        if self.is_truthy(left): return left  # Court-circuit
    else:  # AND
        if not self.is_truthy(left): return left

    return self.evaluate(expr.right)
```

Step 21

Commentaires

Syntaxe:

```
// Ceci est un commentaire
var x = 5;  // Commentaire en fin de ligne
```

Dans le lexer:

```
if self.peek() == '/' and self.peek_next() == '/':
    # Ignorer jusqu'à la fin de ligne
    while self.peek() != '\n' and not self.is_at_end():
        self.advance()
```

Les commentaires sont ignorés au lexing, ils n'atteignent jamais le parser

Built-in Functions

Fonctions natives : implémentées en Python, exposées à Toy

clock() : timestamp actuel (secondes depuis epoch)

```
var start = clock();
// ... code ...
var elapsed = clock() - start;
print elapsed;
```

number() : convertir string → nombre

```
var x = number("123");
print x + 1;  // 124
```

TP #09 - Built-in clock()

Objectif: Implémenter une fonction native `clock()`

À faire:

- Créer classe `BuiltinClock` avec méthode `call(interpreter, arguments)`
  - Retourner `time.time()` (nécessite `import time`)
  - Arity = 0 (pas d'arguments)
- Dans l'interpréteur, au démarrage :
  - `self.globals.define("clock", BuiltinClock())`

TP #09 - Built-in clock()

Dans `test_toy.py` :

```
1   def test_clock_builtin():
2       result = evaluate("clock();")
3       assert isinstance(result, float)
4       assert result > 0
5
6   def test_clock_measures_time(capsys):
7       source = """
8       var start = clock();
9       var end = clock();
10      print end >= start;
11      """
12      interpret(source)
13      captured = capsys.readouterr()
14      assert captured.out == "True\n"
```

Récap Final

Toy est complet

Ce qu’on a construit ensemble

Module 1-2 : Lexer + Parser

- Tokenization
- Recursive descent parsing
- Précédence des opérateurs
- AST

Module 3-4 : Interpreter basics

- Variables avec scoping lexical
- Expressions arithmétiques
- Print statement

Module 5 : Control Flow

- While loops
- For loops (via desugaring)
- Blocks

Module 6 : Functions

- First-class functions
- Closures
- Function calls
- Return statements

Ce qu’on a construit ensemble

Module 7 : ANTLR

- Parser generators
- Grammaires déclaratives
- Visitor pattern

Module 8 : Polishing

- Strings
- Logical operators
- Comments
- Built-in functions

Vous avez créé un langage de programmation complet

Et maintenant ?

Le monde du PL design vous attend

Have fun! 🚀