



Abstract Syntax Tree Generation

Principles of Programming Languages

MEng. Tran Ngoc Bao Duy

Department of Computer Science

Faculty of Computer Science and Engineering

Ho Chi Minh University of Technology, VNU-HCM



Understanding
Abstract Syntax Trees
(AST)

Traversing parse tree

Visitor pattern

Using Visitor in
ANTLR4

① Understanding Abstract Syntax Trees (AST)

② Traversing parse tree

x -> visitX
y -> visitY
program -> visitProgram

③ Visitor pattern

④ Using Visitor in ANTLR4

1. Understanding Abstract Syntax Trees (AST) - Hiểu về Cây Cú Pháp Trừu Tượng (AST)

Abstract Syntax Tree (AST), dịch là Cây Cú Pháp Trừu Tượng, là một cấu trúc dữ liệu dạng cây, trừu tượng hơn so với Parse Tree (Cây Phân tích Cú pháp) mà chúng ta đã thảo luận.

Mục tiêu của AST: AST được tạo ra từ Parse Tree, với mục đích tập trung vào ý nghĩa cú pháp của chương trình, loại bỏ các chi tiết không cần thiết cho các giai đoạn xử lý tiếp theo (như các ký hiệu ngữ pháp trung gian, chi tiết về cú pháp cụ thể).

Trừu tượng hơn Parse Tree như thế nào?

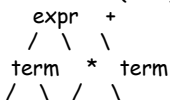
Loại bỏ các nút trung gian không quan trọng: Parse Tree có thể chứa nhiều nút trung gian chỉ để phục vụ quá trình phân tích cú pháp (ví dụ: các non-terminal như `term`, `expr`, ... trong grammar EBNF để xử lý độ ưu tiên). AST thường loại bỏ các nút này và chỉ giữ lại các nút thực sự biểu diễn cấu trúc ngữ nghĩa cốt lõi.

Tập trung vào toán tử và toán hạng: Trong biểu thức, AST thường tập trung vào việc biểu diễn toán tử (ví dụ: `+`, `-`, `*`, `=`) và toán hạng (operands - ví dụ: biến, số, biểu thức con), và quan hệ giữa chúng.

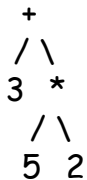
Dạng cây đơn giản hơn, gọn gàng hơn: AST thường có cấu trúc cây đơn giản hơn, phẳng hơn so với Parse Tree. Nó dễ dàng hơn để duyệt và xử lý trong các giai đoạn biên dịch tiếp theo.

Ví dụ (so sánh Parse Tree và AST cho $3 + 5 * 2$):

Parse Tree (Ví dụ): (Có thể phức tạp tùy grammar)



Ví dụ về cây AST: Có cấu trúc đơn giản hơn Parse tree, tập trung vào toán tử và toán hạng.



Trong AST: ** Chúng ta thấy rõ ràng phép toán + là nút gốc, với hai nhánh con là 3 và *. Subtree `*` lại có gốc là toán tử `*` với hai con là `5` và `2`. AST này thể hiện cấu trúc phép toán một cách trực tiếp, dễ hiểu. Các nút trung gian phức tạp của Parse Tree đã được loại bỏ

2. Traversing parse tree - Duyệt Cây Phân tích Cú pháp

Traversing (Duyệt): Để xây dựng AST từ Parse Tree, chúng ta cần duyệt qua Parse Tree.

"Duyệt cây" nghĩa là thăm từng nút của cây theo một thứ tự nhất định.

Mục đích của việc duyệt Parse Tree: Khi duyệt cây, chúng ta sẽ thực hiện các hành động tại mỗi nút để trích xuất thông tin cần thiết từ Parse Tree và xây dựng các nút tương ứng cho AST.

3. Visitor pattern - Mẫu thiết kế Visitor

Visitor Pattern (Mẫu thiết kế Visitor) là một mẫu thiết kế phần mềm (design pattern) rất hữu ích để duyệt các cấu trúc dữ liệu dạng cây (như Parse Tree, AST) và thực hiện các thao tác khác nhau trên các nút của cây một cách linh hoạt và dễ bảo trì.

Ý tưởng của Visitor Pattern:

Tách biệt thuật toán duyệt cây khỏi các thao tác cụ thể trên các nút. Thay vì "cứng nhắc" các thao tác vào chính class của các nút trong cây, Visitor Pattern cho phép bạn định nghĩa các "Visitor" (người thăm) riêng biệt.

Mỗi Visitor class định nghĩa các phương thức (visit methods) cho từng loại nút khác nhau trong cây. Ví dụ, có thể có visitExpr(), visitTerm(), visitFactor(), ...

Khi duyệt cây, tại mỗi nút, chúng ta "chấp nhận" một Visitor. Visitor sẽ gọi phương thức visitXXX() tương ứng với loại nút hiện tại.

Visitor có thể thực hiện bất kỳ thao tác nào khi "thăm" nút: Ví dụ: xây dựng AST, kiểm tra ngữ nghĩa, sinh mã, ...

Ưu điểm của Visitor Pattern:

Tính module hóa (Modularity): Thêm các thao tác mới lên cây dễ dàng bằng cách tạo Visitor mới mà không cần sửa đổi cấu trúc cây.

Tính mở rộng (Extensibility): Dễ dàng mở rộng chức năng bằng cách thêm Visitor classes.

Code sạch và dễ bảo trì: Tách biệt logic duyệt cây và logic xử lý nút, code rõ ràng, dễ hiểu và bảo trì.

4. Using Visitor in ANTLR4 - Sử dụng Visitor trong ANTLR4

ANTLR4 hỗ trợ sẵn Visitor Pattern để duyệt Parse Tree một cách tự động. Khi bạn cấu hình ANTLR4 để generate "Visitor" code, ANTLR4 sẽ tự động sinh ra các interface và classes Visitor tương ứng với grammar của bạn. Các Visitor interface/class sẽ chứa các phương thức visitXXX() tương ứng với mỗi quy tắc parser (rule) trong grammar. Ví dụ, nếu bạn có quy tắc `expr : term '+' term ;`, ANTLR4 sẽ sinh ra phương thức `visitExpr()` trong Visitor interface.

Bạn chỉ cần implement các Visitor interface/class này để định nghĩa các hành động cụ thể bạn muốn thực hiện khi duyệt Parse Tree, ví dụ: xây dựng AST, tính giá trị biểu thức, ...

Cơ chế hoạt động của Visitor trong ANTLR4:

Parser ANTLR4 tạo ra Parse Tree.

Bạn tạo một class kế thừa từ Visitor base class mà ANTLR4 đã sinh ra. Trong class Visitor của bạn, bạn override các phương thức visitXXX() tương ứng với các nút Parse Tree mà bạn muốn xử lý.

Bạn gọi phương thức `accept(yourVisitor)` trên nút gốc của Parse Tree.

ANTLR4 engine sẽ tự động duyệt cây theo chiều sâu (depth-first) và gọi các phương thức visitXXX() tương ứng trên Visitor của bạn khi nó "thăm" các nút Parse Tree.

What is an AST?

Cây Cú Pháp Trừu Tượng (AST) là một biểu diễn dạng cây phân cấp của cấu trúc cú pháp trừu tượng của mã nguồn. Không giống như cây phân tích cú pháp, AST loại bỏ các chi tiết cú pháp không cần thiết và tập trung vào cấu trúc thiết yếu của chương trình.

thể hiện cấu trúc phân cấp lược bỏ chi tiết ko cần thiết

Definition

giữ lại chi tiết về mặt ngữ nghĩa

An **Abstract Syntax Tree (AST)** is a hierarchical tree representation of the abstract syntactic structure of source code. Unlike parse trees, ASTs omit unnecessary syntactic details and focus on the essential structure of the program.

COMPARISON OF PARSE TREE AND AST

Aspect	Parse Tree	AST
Includes syntax Structure Purpose	Yes More verbose Parsing and verification	No Simplified Semantic analysis and code generation

Cũng là cấu trúc cây phân cấp như Parse tree.

Không quan tâm mọi chi tiết cụ thể của cú pháp, mà chỉ tập trung vào bản chất cấu trúc ngữ nghĩa.

Khác với Parse tree ở chỗ nó loại bỏ các chi tiết cú pháp không cần thiết (như các kí hiệu trung gian (non-terminal), dấu phân cách...).

Tập trung vào cấu trúc thiết yếu.



Giải thích bảng so sánh:

Aspect: Includes syntax (Bao gồm cú pháp)

Parse Tree: Yes (Có) - Parse Tree bao gồm toàn bộ cú pháp của mã nguồn. Nó phản ánh chính xác cấu trúc grammar, bao gồm cả các chi tiết về cú pháp cụ thể (từ khóa, dấu câu, ký hiệu ngữ pháp trung gian). Parse Tree thường còn được gọi là Concrete Syntax Tree (CST) vì nó thể hiện cú pháp một cách "cụ thể" và đầy đủ.

AST: No (Không) - AST không bao gồm mọi chi tiết cú pháp. Nó loại bỏ các chi tiết không cần thiết cho ngữ nghĩa. AST chỉ giữ lại những thành phần cốt lõi về mặt cấu trúc và ý nghĩa.

Aspect: Structure (Cấu trúc)

Parse Tree: More verbose (Chi tiết hơn) - Parse Tree thường có cấu trúc dài dòng hơn, phức tạp hơn ("verbose"). Nó có thể có nhiều tầng lớp nút, nhiều nút trung gian chỉ để khớp với grammar.

AST: Simplified (Đơn giản hóa) - AST có cấu trúc đơn giản hơn, gọn gàng hơn ("simplified"). Nó được tối ưu hóa để dễ dàng xử lý và phân tích ngữ nghĩa. AST thường phẳng hơn, ít nút trung gian hơn.

Aspect: Purpose (Mục đích)

Parse Tree: Parsing and verification (Phân tích cú pháp và xác minh) - Mục đích chính của Parse Tree là chứng minh rằng mã nguồn đầu vào là hợp lệ về mặt cú pháp theo grammar (quá trình parsing), và cung cấp một cấu trúc chi tiết cho giai đoạn tiếp theo.

AST: Semantic analysis and code generation (Phân tích ngữ nghĩa và sinh mã) - Mục đích chính của AST là làm đầu vào cho các giai đoạn biên dịch tiếp theo, như phân tích ngữ nghĩa (semantic analysis) (kiểm tra kiểu, kiểm tra biến đã khai báo, ...) và sinh mã (code generation) (tạo ra mã máy, mã bytecode, ...). AST cung cấp một cấu trúc trừu tượng, tập trung vào ý nghĩa, phù hợp cho các bước xử lý ngữ nghĩa và sinh mã.



TRAVERSING PARSE TREE GENERATED BY ANTLR4

Generated parse tree



Suppose that in the BKOOL.g4 file we write as follows:

```
1 grammar BKOOL;
2
3 program: expr EOF; // starting rule
4 expr: expr ADD term | expr SUB term | term;
5 term: term MUL fact | term DIV fact | fact;
6 fact: LP expr RP | ID | INT;
7
8 ADD: '+'; SUB: '-'; MUL: '*'; DIV: '/';
9 ID: [a-zA-Z]+; INT: [0-9]+;
10 WS: [ \t\r\n]+ -> skip;
```

Generated parse tree



Suppose that in the BKOOL.g4 file we write as follows:

```
1 grammar BKOOL;
2
3 program: expr EOF; // starting rule
4 expr: expr ADD term | expr SUB term | term;
5 term: term MUL fact | term DIV fact | fact;
6 fact: LP expr RP | ID | INT;
7
8 ADD: '+'; SUB: '-'; MUL: '*'; DIV: '/';
9 ID: [a-zA-Z]+; INT: [0-9]+;
10 WS: [ \t\r\n]+ -> skip;
```

Generated parse tree

File `BK00LLexer.py` chứa code Python cho lexer (scanner). Lexer này được sinh tự động từ các lexer rules mà chúng ta đã định nghĩa trong `BK00L.g4` (phần viết HOA như `ADD`, `ID`, `INT`, `WS`, `LP`, `RP`). Code lexer được tổ chức trong một class tên `BK00LLexer`. Class này chứa các phương thức để phân tích mã nguồn đầu vào thành chuỗi token.

Then, when we run the command `python run.py gen`, we see that 7 files are generated in the `target` directory, with a focus on the following 2 files:

- 1 `BK00LLexer.py`: containing the lexer (scanner),
in class `BK00LLexer`
- 2 `BK00LParser.py`: containing the parser, in class `BK00LParser`

File `BK00LParser.py` chứa code Python cho parser. Parser này được sinh tự động từ các parser rules trong `BK00L.g4` (phần viết thường như `program`, `expr`, `term`, `fact`). Code parser được tổ chức trong một class tên `BK00LParser`. Class này chứa các phương thức để phân tích cú pháp chuỗi token (từ lexer) và tạo ra Parse Tree.



Understanding ANTLR4 parse tree

To traverse the generated parse tree, we need to understand the data types of the nodes in the tree according to the following rules:

- `grammar BK00L` generates `class BK00LParser`. The parser class is responsible for constructing parse trees according to the grammar rules.
- The nested `*Context` classes inside `BK00LParser` represent nodes in the parse tree and help manage rule-specific parsing:
 - `ProgramContext` from parser rule `program`.
 - `ExprContext` from parser rule `expr`. **vd: kiểu dữ liệu `BK00LParserExprContext` viết hoa chữ cái đầu tiên**
 - `TermContext` from parser rule `term`.
 - `FactContext` from parser rule `fact`.



Để có thể duyệt (traverse) và xử lý cây phân tích cú pháp mà ANTLR4 tạo ra, bước đầu tiên là chúng ta cần hiểu rõ kiểu dữ liệu của các nút trong cây đó. Slide này sẽ giải thích quy tắc đặt tên và cấu trúc của các nút cây.

"grammar BKOOL generates class BKOOLParser": Khi bạn sử dụng ANTLR4 để xử lý file grammar BKOOL.g4, một trong những file code Python được sinh ra là BKOOLParser.py. File này chứa một class tên BKOOLParser. Tên class parser được tự động tạo ra dựa trên tên grammar (trong trường hợp này là BKOOL) và thêm hậu tố Parser.

"The parser class is responsible for constructing parse trees according to the grammar rules.": Class BKOOLParser chính là class parser. Chức năng chính của class này là sử dụng các quy tắc grammar mà bạn đã định nghĩa trong BKOOL.g4 để phân tích cú pháp mã nguồn đầu vào và tạo ra cây phân tích cú pháp (parse tree).

"The nested *Context classes inside BKOOLParser represent nodes in the parse tree and help manage rule-specific parsing:"

"The nested *Context classes inside BKOOLParser...": Bên trong class BKOOLParser (trong file BKOOLParser.py), ANTLR4 tự động sinh ra các class con (nested classes) với tên kết thúc bằng Context.

"...represent nodes in the parse tree...": Các class *Context này chính là kiểu dữ liệu đại diện cho các nút (nodes) trong cây phân tích cú pháp. Mỗi instance (đối tượng) của một class *Context là một nút trong parse tree.

"...and help manage rule-specific parsing": Các class *Context không chỉ đơn thuần là kiểu dữ liệu nút. Chúng còn chứa code và phương thức liên quan đến việc phân tích cú pháp cho từng quy tắc grammar cụ thể. Mỗi class *Context gắn liền với một quy tắc parser trong grammar.

Danh sách các *Context classes cụ thể (sinh ra từ grammar BKOOL.g4):

"ProgramContext from parser rule program.": Class ProgramContext đại diện cho các nút trong parse tree được tạo ra khi parser áp dụng quy tắc program : expr EOF ;. Nói cách khác, mỗi nút ProgramContext trong cây tương ứng với một instance của quy tắc program.

"ExprContext from parser rule expr.": Class ExprContext đại diện cho các nút được tạo ra khi parser áp dụng quy tắc expr.

"TermContext from parser rule term.": Class TermContext đại diện cho các nút được tạo ra khi parser áp dụng quy tắc term.

"FactContext from parser rule fact.": Class FactContext đại diện cho các nút được tạo ra khi parser áp dụng quy tắc fact.

Tóm lại, slide "Understanding ANTLR4 parse tree" giải thích:

ANTLR4 sinh ra class BKOOLParser (tên tùy thuộc grammar) chứa parser.

Bên trong BKOOLParser, ANTLR4 sinh ra các class *Context (ví dụ: ProgramContext, ExprContext, TermContext, FactContext).

Mỗi *Context class đại diện cho kiểu dữ liệu của các nút trong parse tree tương ứng với một quy tắc parser trong grammar. Khi parser áp dụng một quy tắc, nó sẽ tạo ra một nút có kiểu là class *Context tương ứng.

Để duyệt và xử lý parse tree, chúng ta sẽ làm việc chủ yếu với các đối tượng thuộc các class *Context này. Chúng cung cấp thông tin về nút trong cây và các phương thức để truy cập các nút con của chúng.

Common Ways to Access Children

Parse trees are represented by context classes, and you can access child nodes from the parent node using various methods provided by ANTLR's `ParserRuleContext` class. Assume that `ctx` is storing a node in the parse tree:

- 1 **By Indexing:** `ctx.getChild(index)` => lấy dc node con
- 2 **By Type** (Specific Rule Context): `ctx.term()`, in which `term` is a rule type in the **right hand side** of a parser rule. The return value could be in:
 - A single object – when the grammar rule allows only one occurrence of that child.
 - A list of objects – when the rule allows multiple occurrences of that child (includes rules in EBNF).
- 3 **By Field Reference:** `ctx.term(0)` or `ctx.term(1)`

Other supported methods in `ParserRuleContext` class:

- Use `getChildCount()` to get the number of children.
- Use `getText()` to get the string representation of a node.



Giải thích chi tiết từng phương pháp truy cập con:

"Parse trees are represented by context classes...": Nhắc lại rằng các nút trong Parse Tree được biểu diễn bằng các class `*Context` (ví dụ: `ProgramContext`, `ExprContext`, `TermContext`, `FactContext`).

"...you can access child nodes from the parent node using various methods provided by ANTLR's `ParserRuleContext` class.": **Tất cả các class `*Context` này đều kế thừa từ class `ParserRuleContext` trong thư viện ANTLR runtime. `ParserRuleContext` cung cấp một loạt các phương thức để truy cập đến các nút con từ một nút cha.** Slide này giới thiệu các cách phổ biến nhất. Giả sử chúng ta có một biến `ctx` đang tham chiếu đến một đối tượng `*Context` (nút cha trong Parse Tree).

1 By Indexing: ctx.getChild(index)

Tạm dịch: "1 Bảng Chỉ Mục: ctx.getChild(index)"

Giải thích:

ctx.getChild(index): Phương thức này cho phép bạn truy cập nút con tại một vị trí cụ thể trong danh sách các nút con của ctx, sử dụng chỉ mục (index) bắt đầu từ 0.

index: Là một số nguyên (integer) từ 0 đến getChildCount() - 1.

Trả về: Phương thức getChild(index) trả về một đối tượng ParseTree (hoặc một subclass của ParseTree, thường là một *Context object). Nếu index không hợp lệ (ngoài phạm vi), có thể gây ra lỗi.

Ví dụ (dựa trên grammar BKOOL.g4): Giả sử bạn có một nút exprContext kiểu ExprContext đại diện cho quy tắc `expr : expr ADD term`. Trong Parse Tree, nút exprContext này có thể có 3 nút con: expr (con thứ nhất, index 0), ADD (con thứ hai, index 1), và term (con thứ ba, index 2). Bạn có thể truy cập chúng như sau:

left_expr_child = exprContext.getChild(0)	# Lấy nút con `expr` bên trái
add_op_child = exprContext.getChild(1)	# Lấy nút con `ADD` (token '+')
term_child = exprContext.getChild(2)	# Lấy nút con `term` bên phải

2 By Type (Specific Rule Context): `ctx.term()`

Tạm dịch: "2 Bằng Kiểu (Context Quy Tắc Cụ Thể): `ctx.term()`, trong đó `term` là một kiểu quy tắc ở vế phải của một quy tắc parser. Giá trị trả về có thể là:"

Giải thích:

`ctx.term()`: Đây là một cách tiện lợi và type-safe hơn để truy cập nút con, dựa trên tên của quy tắc grammar mà nút con đó được tạo ra từ. Nếu bạn biết rằng một nút con của `ctx` được tạo ra từ quy tắc `term` trong grammar, bạn có thể gọi phương thức `ctx.term()` (với tên quy tắc `term` làm tên phương thức).

`term` is a rule type in the right hand side of a parser rule. - Tên phương thức (`term()` trong ví dụ) phải trùng với tên của một quy tắc parser (viết thường) được sử dụng ở vế phải của quy tắc hiện tại (quy tắc mà `ctx` đại diện). Ví dụ, nếu quy tắc là `expr : expr ADD term`, thì `term` là một rule type ở vế phải.

Return value (Giá trị trả về): Kiểu trả về của `ctx.term()` có thể khác nhau tùy thuộc vào số lần quy tắc `term` xuất hiện trong quy tắc grammar định nghĩa `ctx`.

- A single object – when the grammar rule allows only one occurrence of that child.

Tạm dịch: "• Một đối tượng duy nhất – khi quy tắc grammar chỉ cho phép một lần xuất hiện của con đó."

Nếu quy tắc grammar định nghĩa rằng nút con kiểu `term` xuất hiện duy nhất một lần (hoặc tối đa một lần), phương thức `ctx.term()` sẽ trả về một đối tượng duy nhất kiểu `TermContext` (hoặc `None` nếu không có).

Ví dụ (dựa trên grammar `BK00L.g4`): Trong quy tắc `fact : LP expr RP ;`, quy tắc `expr` xuất hiện một lần giữa `LP` và `RP`. Do đó, nếu `factContext` là một đối tượng `FactContext` cho quy tắc này, bạn có thể gọi:

`expr_child = factContext.expr()` # Trả về một đối tượng `ExprContext` (hoặc `None`)

`expr_child` sẽ là nút con kiểu `ExprContext` đại diện cho `expr` bên trong dấu ngoặc.

- A list of objects – when the rule allows multiple occurrences of that child (includes rules in EBNF).

Tạm dịch: "• Một danh sách các đối tượng – khi quy tắc cho phép nhiều lần xuất hiện của con đó (bao gồm các quy tắc trong EBNF)."

Nếu quy tắc grammar định nghĩa rằng nút con kiểu `term` có thể xuất hiện nhiều lần (ví dụ sử dụng toán tử lặp `*`, `+`, `?` trong EBNF), phương thức `ctx.term()` sẽ trả về một danh sách (list) các đối tượng kiểu `TermContext`. Danh sách này có thể rỗng nếu không có nút con nào kiểu `term`.

Ví dụ (giả sử có quy tắc): `functionCall : ID LP (expr ',' expr)* ? RP ;` (gọi hàm với danh sách tham số tùy chọn, phân tách bằng dấu phẩy). Trong quy tắc này, `expr` có thể xuất hiện nhiều lần (0 hoặc nhiều) trong phần `(expr ',' expr)* ?`. Nếu `functionCallContext` là một đối tượng `FunctionCallContext`, thì:

`arg_expr_list = functionCallContext.expr()` # Trả về một list các đối tượng `ExprContext` (có thể rỗng)

`arg_expr_list` sẽ là danh sách các nút con kiểu `ExprContext`, mỗi nút đại diện cho một biểu thức tham số trong lời gọi hàm.

3 By Field Reference: `ctx.term(0)` or `ctx.term(1)`

Tạm dịch: "3 Bằng Tham Chiếu Trường: `ctx.term(0)` hoặc `ctx.term(1)`"

Giải thích:

`ctx.term(0)`, `ctx.term(1)`, ...: Đây là cách truy cập nút con khi có nhiều nút con cùng kiểu (cùng được tạo ra từ cùng một quy tắc grammar) trong một nút cha, và bạn muốn lấy một nút con cụ thể theo thứ tự xuất hiện của chúng trong grammar rule.

`ctx.term(index)`: Bạn gọi phương thức `ctx.term(index)` với một chỉ mục số nguyên `index`. Chỉ mục này không phải là chỉ mục vị trí trong danh sách tất cả các nút con (như `getChild(index)`), mà là chỉ mục thứ tự xuất hiện của các nút con kiểu `term` trong grammar rule. Chỉ mục cũng bắt đầu từ 0.

Trả về: `ctx.term(index)` trả về một đối tượng `TermContext` (hoặc `None` nếu không có nút con `term` nào ở vị trí chỉ mục đó).

Ví dụ (dựa trên grammar `BKOOOL.g4`): Trong quy tắc `expr : expr ADD term`, có một `expr` và một `term` ở vế phải. Tuy nhiên, nếu grammar có dạng: `binaryOp : expr operator expr`; (ví dụ, `operator` có thể là `+`, `-`, `*`, `/`). Trong trường hợp này, quy tắc `binaryOp` có thể được hiểu là có hai nút con kiểu `expr`: `expr` thứ nhất (vế trái) và `expr` thứ hai (vế phải). Bạn có thể truy cập chúng như:

`left_expr = binaryOpContext.expr(0)` # Lấy nút con `expr` thứ nhất (vế trái)

`right_expr = binaryOpContext.expr(1)` # Lấy nút con `expr` thứ hai (vế phải)

Ở đây, `binaryOpContext.expr(0)` lấy `expr` đầu tiên xuất hiện trong quy tắc `binaryOp : expr operator expr`, và `binaryOpContext.expr(1)` lấy `expr` thứ hai.

binaryOpContext.expr(1) lấy expr thứ hai.

Other supported methods in ParserRuleContext class:

"• Use getChildCount() to get the number of children."

Tạm dịch: "• Sử dụng getChildCount() để lấy số lượng nút con."

ctx.getChildCount(): Phương thức này trả về một số nguyên, là tổng số nút con mà nút ctx có. Nó hữu ích khi bạn muốn duyệt tất cả các nút con một cách tuần tự, hoặc kiểm tra xem một nút có bao nhiêu con.

"• Use getText() to get the string representation of a node."

Tạm dịch: "• Sử dụng getText() để lấy biểu diễn chuỗi của một nút."

ctx.getText(): Phương thức này trả về một chuỗi (string), là chuỗi văn bản (text) tương ứng với nút ctx và tất cả các nút con cháu của nó trong mã nguồn gốc. Ví dụ, nếu ctx là nút expr cho biểu thức $5 * 2$, thì ctx.getText() có thể trả về chuỗi "5*2"

Tóm lại, slide "Common Ways to Access Children" giới thiệu các phương pháp chính để duyệt Parse Tree trong ANTLR4:

getChild(index): Truy cập con theo vị trí index.

ctx.ruleName(): Truy cập con theo kiểu (tên quy tắc grammar). Trả về một đối tượng hoặc một danh sách tùy thuộc vào grammar.

ctx.ruleName(index): Truy cập con theo kiểu và thứ tự xuất hiện (field reference).

getChildCount(): Lấy số lượng con.

getText(): Lấy văn bản của nút và các con cháu.

Giả sử: Chúng ta có một nút cha `exprContext` đại diện cho một biểu thức $b + c + d$ trong parse tree.

1. `ctx.getChild(index)` (Truy cập bằng Chỉ Mục):

Cách hoạt động: Giống như bạn truy cập phần tử trong một mảng bằng số thứ tự (`index`). Bạn cần biết vị trí chính xác của nút con mà bạn muốn lấy trong danh sách các nút con của `ctx`.

Ví dụ: Để lấy nút con thứ hai (dấu + đầu tiên trong $b + c + d$) của `exprContext`, bạn dùng:

`dau_cong_con = exprContext.getChild(1) # Index = 1 (thứ hai trong danh sách con)`

Điểm khác biệt:

Trả về chung chung: Trả về kiểu chung chung `ParseTree` (hoặc `TerminalNode` cho token, `RuleContext` cho rule), bạn có thể cần kiểm tra kiểu hoặc cast về kiểu cụ thể (ví dụ `TerminalNode` nếu bạn biết con là token).

Ít type-safe: Dễ bị lỗi nếu bạn đếm sai index hoặc cấu trúc cây thay đổi.

Thích hợp khi: Bạn cần duyệt qua tất cả các nút con một cách tuần tự mà không quan tâm đến kiểu cụ thể của chúng, hoặc khi bạn biết chắc chắn vị trí của nút con mình muốn lấy.

2. `ctx.ruleName()` (Truy cập bằng Kiểu - Tên Quy Tắc):

Cách hoạt động: Dùng tên quy tắc grammar (viết thường) như một phương thức để truy cập trực tiếp nút con theo kiểu của nó. ANTLR4 tự động tạo ra các phương thức này dựa trên grammar.

Ví dụ: Để lấy nút con kiểu `term` (ví dụ, trong quy tắc `expr : expr ADD term`), bạn dùng:

`nut_term_con = exprContext.term() # Gọi phương thức `term()``

Điểm khác biệt:

Dựa vào kiểu: Quan trọng là kiểu của nút con (tên quy tắc grammar).

Type-safe: Trả về kiểu `*Context` cụ thể (ví dụ `TermContext`), giúp code rõ ràng và ít lỗi hơn.

Trả về một hoặc danh sách:

Một đối tượng: Nếu quy tắc grammar chỉ cho phép một nút con kiểu `ruleName` (hoặc tối đa một).

Danh sách: Nếu quy tắc grammar cho phép nhiều nút con kiểu `ruleName` (ví dụ dùng `*`, `+`, `?` trong EBNF).

Thích hợp khi: Bạn muốn truy cập nút con có kiểu cụ thể và bạn biết rằng có duy nhất một hoặc một danh sách các con kiểu đó.

3. ctx.ruleName(index) (Truy cập bằng Tham Chiếu Trường - Kiểu và Index):

Cách hoạt động: Kết hợp cả kiểu và index khi có nhiều nút con cùng kiểu. Dùng tên quy tắc làm phương thức, và truyền vào index để chỉ định nút con thứ mấy (theo thứ tự xuất hiện trong grammar rule) mà bạn muốn lấy.

Ví dụ: Giả sử (ví dụ này hơi giả định, vì grammar BKOOL.g4 không có trường hợp này trong expr, nhưng để minh họa): nếu quy tắc expr được sửa thành `expr : term ADD expr SUB term ;` (có cả term và expr xuất hiện nhiều lần). Để lấy term thứ nhất và expr thứ hai, bạn dùng:

`nut_term_con_thu_nhat = exprContext.term(0) # Lấy term thứ nhất (index 0)`

`nut_expr_con_thu_hai = exprContext.expr(1) # Lấy expr thứ hai (index 1)`

Điểm khác biệt:

Dùng khi nhiều con cùng kiểu: Dùng khi một nút cha có nhiều nút con được sinh ra từ cùng một quy tắc grammar.

Chỉ định thứ tự: Index ở đây là thứ tự xuất hiện của các nút con cùng kiểu trong định nghĩa grammar rule, không phải index vị trí trong tất cả các con.

Type-safe: Trả về kiểu `*Context` cụ thể.

Thích hợp khi: Bạn có nhiều nút con cùng kiểu và cần truy cập chúng theo thứ tự cụ thể như được định nghĩa trong grammar.

Traverse the parse tree

ANTLR4 provides two ways to traverse the parse tree, categorized into:

- ① **Implicit Traversal** (Listener Pattern) duyệt cây ko tường minh
- ② **Explicit Traversal** (Visitor Pattern) duyệt cây tường minh



Traverse the parse tree

ANTLR4 provides two ways to traverse the parse tree, categorized into:

Tự động duyệt cây theo chiều sâu (depth-first), cụ thể là theo thứ tự hậu thứ tự (post-order). Bạn không cần viết code để điều khiển thứ tự duyệt cây, ANTLR4 runtime sẽ tự động thực hiện việc duyệt cây.

- 1 **Implicit Traversal** (Listener Pattern): automatically walks through the parse tree depth-first (post-order traversal). The listener methods are triggered automatically when entering and exiting parse tree nodes.

- 2 **Explicit Traversal** (Visitor Pattern)

COMPARISON OF VISITOR AND LISTENER IN ANTLR4

Feature	Visitor	Listener
Control	Manual control over traversal	Automatic DFS traversal
Method Names	visitXXX(ctx)	enterXXX(ctx), exitXXX(ctx)
Flexibility	High	Moderate
Ease of Use	Complex	Simple
Use Case	Evaluation, transformation	Analysis, validation
Performance	Slightly slower	Faster





"provides manual control over how nodes are visited in the parse tree...": Cung cấp khả năng kiểm soát thủ công hoàn toàn cách duyệt cây. Bạn chủ động viết code để điều khiển thứ tự duyệt, bạn quyết định nút nào được thăm, nút nào bỏ qua, duyệt theo thứ tự nào (không bị giới hạn ở depth-first hay post-order).

"...allowing selective traversal of child nodes.": Cho phép duyệt chọn lọc các nút con. Bạn có thể quyết định, tại một nút nào đó, có duyệt tiếp các nút con của nó hay không, hoặc chỉ duyệt một số nút con nhất định.

ANTLR4 provides two ways to traverse the parse tree, categorized into:

- 1 **Implicit Traversal** (Listener Pattern)
- 2 **Explicit Traversal** (Visitor Pattern) : provides manual control over how nodes are visited in the parse tree, allowing selective traversal of child nodes. **điều chỉnh vào ra node thủ công**

COMPARISON OF VISITOR AND LISTENER IN ANTLR4

Feature	Visitor	Listener
Control	Manual control over traversal	Automatic DFS traversal
Method Names	visitXXX(ctx)	enterXXX(ctx), exitXXX(ctx)
Flexibility	High	Moderate
Ease of Use	Complex	Simple
Use Case	Evaluation, transformation	Analysis, validation
Performance	Slightly slower	Faster

For generating AST (Transformation), choosing **VISITOR**.

Do AST generation thường yêu cầu kiểm soát linh hoạt quá trình duyệt cây và biến đổi các nút, Visitor Pattern với khả năng kiểm soát thủ công và trả về giá trị sẽ là công cụ mạnh mẽ hơn so với Listener.



VISITOR PATTERN

Story: J98 – The Musical Wanderer

J98 was a traveler in the world of music, unbound by place or rules. He wandered through life, carrying his melodies, meeting different people, and leaving a lasting impact with each encounter.

- ① Along his path, **J98** met **Bé La**, a child to whom he gave **3.5 million VND** without explanation.
- ② **J98** crossed paths with **K-IBM**, a powerful figure in the industry. Without hesitation, he confronted **K-IBM**, accusing him of past oppression.
- ③ **J98** arrived at **Sâu Phát Sáng**, his loyal fan club. He sang “**Giông Bão**”, a song of resilience, as a tribute to those who had stood by him.

If in the world of object-oriented programming, how should we define objects (associated with behaviors)?



Slide này dùng câu chuyện "J98 - The Musical Wanderer" để dễ hình dung về Visitor Pattern.

Câu chuyện J98 (Ví von cho Visitor Pattern):

J98: Một nghệ sĩ du mục, đi khắp nơi, gặp gỡ nhiều người và để lại ấn tượng qua mỗi cuộc gặp. (Ví von cho Visitor - người "thăm" các đối tượng khác nhau).

Các nhân vật J98 gặp (Bé La, K-IBM, Sâu Phát Sáng): Những người khác nhau với các phản ứng khác nhau khi gặp J98. (Ví von cho các loại nút khác nhau trong Parse Tree - ProgramContext, ExprContext, TermContext, ... hoặc các đối tượng khác nhau trong một cấu trúc phức tạp).

Bé La: Nhận tiền (3.5 triệu VND) từ J98. (Ví von cho một loại nút/đối tượng có hành động/xử lý cụ thể khi được "thăm").

K-IBM: Bị J98 chỉ trích. (Ví von cho một loại nút/đối tượng khác có hành động/xử lý khác khi được "thăm").

Sâu Phát Sáng (Fan club): Được J98 hát tặng. (Ví von cho một loại nút/đối tượng khác nữa với hành động/xử lý riêng).

Vấn đề đặt ra (trong lập trình OOP): Nếu có nhiều loại "người" (đối tượng - Bé La, K-IBM, Sâu Phát Sáng) và bạn muốn J98 (Visitor) có hành động khác nhau tùy thuộc vào "người" mà J98 gặp, làm sao để code OOP gọn gàng và dễ mở rộng?

Story: J98 – The Musical Wanderer

J98 was a traveler in the world of music, unbound by place or rules. He wandered through life, carrying his melodies, meeting different people, and leaving a lasting impact with each encounter.

- ① Along his path, **J98** met **Bé La**, a child to whom he gave **3.5 million VND** without explanation.
- ② **J98** crossed paths with **K-IBM**, a powerful figure in the industry. Without hesitation, he confronted **K-IBM**, accusing him of past oppression.
- ③ **J98** arrived at **Sâu Phát Sáng**, his loyal fan club. He sang **“Giông Bão”**, a song of resilience, as a tribute to those who had stood by him.

If in the world of object-oriented programming, how should we define objects (associated with behaviors)?

Requirement: Implement these classes in Programming Code - Exercise 1 with the support of type() function.



Implementation in OOP (1)

```
1 class Person: pass
2 class BeLa(Person): pass
3 class KIBM(Person): pass
4 class SauPhatSang(Person): pass
5
6 class Visitor: pass
7 class J98(Visitor):
8     def visit(self, ctx: Person):
9         if type(ctx) is BeLa:
10             print("I will give you 3.5 million VND")
11         elif type(ctx) is KIBM:
12             print("You oppressed me in the past")
13         elif type(ctx) is SauPhatSang:
14             print("I will sing a song for you")
```

Cách 1: Hàm visit lớn duy nhất với if-elif-else (Code "Implementation in OOP (1)")

Mô tả: Class J98 có một hàm visit(ctx: Person) duy nhất. Bên trong dùng if type(ctx) is ... để kiểm tra kiểu của "người" và thực hiện hành động tương ứng.

Vấn đề:

Khó mở rộng: Nếu có thêm loại "người" mới (ví dụ "Ông Ba"), bạn phải sửa code hàm visit, thêm elif type(ctx) is ÔngBa: Hàm visit ngày càng dài và rối.

Không chuyên nghiệp: Một hàm visit quá lớn xử lý mọi thứ, vi phạm nguyên tắc single responsibility (mỗi class/method nên có một trách nhiệm duy nhất) và open/closed principle (mở cho mở rộng, đóng cho sửa đổi).



Implementation in OOP (1)

```
1 class Person: pass
2 class BeLa(Person): pass
3 class KIBM(Person): pass
4 class SauPhatSang(Person): pass
5
6 class Visitor: pass
7 class J98(Visitor):
8     def visit(self, ctx: Person):
9         if type(ctx) is BeLa:
10             print("I will give you 3.5 million VND")
11         elif type(ctx) is KIBM:
12             print("You oppressed me in the past")
13         elif type(ctx) is SauPhatSang:
14             print("I will sing a song for you")
```

Some issues:

- If there are more than three concrete subclasses of `Person`, what will the body of the `visit` method look like?
- In reality, handling an object belonging to a concrete class is not that simple. Would implementing everything within a single method be considered professional in terms of implementation?





From simple implementation to visitor pattern (2)

Divide the `visit` method into smaller visit methods:

```

1 class Visitor: pass
2 class J98(Visitor):
3     def visit(self, ctx: Person):
4         if type(ctx) is BeLa:
5             return self.visitBeLa(ctx)
6         elif type(ctx) is KIBM:
7             return self.visitKIBM(ctx)
8         elif type(ctx) is SauPhatSang:
9             return self.visitSauPhatSang(ctx)
10    def visitBeLa(self, ctx: BeLa):
11        print("I will give you 3.5 million VND")
12    def visitKIBM(self, ctx: KIBM):
13        print("You oppressed me in the past")
14    def visitSauPhatSang(self, ctx: SauPhatSang):
15        print("I will sing a song for you")

```

Cách 2: Chia visit thành các hàm visitXXX nhỏ hơn (Code "From simple implementation to visitor pattern (2)")

Mô tả: Class J98 vẫn có hàm visit(ctx: Person) chính, nhưng bên trong hàm này chỉ điều phối đến các hàm visitBeLa(ctx), visitKIBM(ctx), visitSauPhatSang(ctx) riêng biệt, mỗi hàm xử lý một loại "người".

Cải thiện: Code có cấu trúc hơn, dễ đọc hơn, mỗi hàm visitXXX nhỏ gọn hơn.

Vấn còn vấn đề: Hàm visit chính vẫn dùng if-elif-else để kiểm tra kiểu và gọi hàm visitXXX tương ứng. Vẫn phải sửa hàm visit chính khi có loại "người" mới. Vẫn chưa thực sự "generalized" (tổng quát hóa) và "professional".

From simple implementation to visitor pattern (2)

Divide the `visit` method into smaller visit methods:

```
1 class Visitor: pass
2 class J98(Visitor):
3     def visit(self, ctx: Person):
4         if type(ctx) is BeLa:
5             return self.visitBeLa(ctx)
6         elif type(ctx) is KIBM:
7             return self.visitKIBM(ctx)
8         elif type(ctx) is SauPhatSang:
9             return self.visitSauPhatSang(ctx)
10    def visitBeLa(self, ctx: BeLa):
11        print("I will give you 3.5 million VND")
12    def visitKIBM(self, ctx: KIBM):
13        print("You oppressed me in the past")
14    def visitSauPhatSang(self, ctx: SauPhatSang):
15        print("I will sing a song for you")
```

Issue: Even though it has been divided into smaller visit methods, the parent `visit` function is still not professional and generalized; each time a new object type is introduced, it must be explicitly defined.



From simple implementation to visitor pattern (3)



Divide the `visit` method into smaller visit methods:

```
1 class Visitor: pass
2 class J98(Visitor):
3     def visit(self, ctx: Person):
4         class_name = ctx.__class__.__name__
5         visit = getattr(self, "visit" + class_name)
6         return visit(ctx)
7     def visitBeLa(self, ctx: BeLa):
8         print("I will give you 3.5 million VND")
9     def visitKIBM(self, ctx: KIBM):
10        print("You oppressed me in the past")
11    def visitSauPhatSang(self, ctx: SauPhatSang):
12        print("I will sing a song for you")
```

Cách 3: Dùng `getattr` để gọi hàm `visitXXX` động (Code "From simple implementation to visitor pattern (3)")

Mô tả: Hàm `visit(ctx: Person)` chính không dùng `if-elif-else` nữa. Thay vào đó, dùng `getattr(self, "visit_" + class_name)` để tự động tìm và gọi hàm `visitXXX` dựa trên tên class của đối tượng `ctx`.

Cải thiện: Hàm `visit` chính ngắn gọn hơn, không còn `if-elif-else`. Về lý thuyết, không cần sửa hàm `visit` chính khi thêm loại "người" mới (nếu đặt tên class và hàm `visitXXX` theo quy ước).

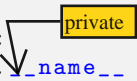
Vẫn còn vấn đề (theo slide): "Awkward and unprofessional", "failing to demonstrate encapsulation". Dùng `getattr` để gọi hàm theo tên chuỗi có thể coi là kém tường minh, khó debug, và "phá vỡ" tính đóng gói (encapsulation) vì logic điều khiển luồng thực thi dựa vào tên chuỗi chứ không phải kiểu đối tượng một cách tường minh. Trong ngữ cảnh này, slide có vẻ muốn nhấn mạnh rằng cách này vẫn chưa phải là giải pháp OOP "chuẩn mực" nhất cho vấn đề này. (Lưu ý: `getattr` có những ứng dụng hợp lệ trong dynamic programming, nhưng ở đây slide đang muốn dẫn dắt đến Visitor Pattern).

From simple implementation to visitor pattern (3)



Divide the `visit` method into smaller visit methods:

```
1 class Visitor: pass
2 class J98(Visitor):
3     def visit(self, ctx: Person):
4         class_name = ctx.__class__.__name__
5         visit = getattr(self, "visit" + class_name)
6         return visit(ctx)
7     def visitBeLa(self, ctx: BeLa):
8         print("I will give you 3.5 million VND")
9     def visitKIBM(self, ctx: KIBM):
10        print("You oppressed me in the past")
11    def visitSauPhatSang(self, ctx: SauPhatSang):
12        print("I will sing a song for you")
```



Issue: The implementation of the parent `visit` method is extremely awkward and unprofessional, failing to demonstrate encapsulation.

From simple implementation to visitor pattern (4)

Implement Exercise 2 in Programming Code:

Follow the steps below to modify the existing J98's story implementation:

- 1 In each concrete class of `Person`, define an additional `accept` method that takes a `Visitor` object as a parameter, calls, and returns the corresponding `visit` method.
Example: In the `BeLa` class, call and return the `visitBeLa` method.
- 2 In the `visit` method of `J98`, call and return the `accept` method.
- 3 Keep all previously defined `visit` methods in the `J98` class unchanged.



Slide đưa ra 3 bước để biến code cũ thành code theo Visitor Pattern:

Bước 1: Thêm phương thức accept vào các class Person cụ thể (BeLa, KIBM, SauPhatSang).

```
class BeLa(Person):
```

```
    def accept(self, v: Visitor): # Nhận Visitor `v`
```

```
        return v.visitBeLa(self) # Gọi v.visitBeLa và truyền đối tượng BeLa (self)
```

Ý nghĩa của accept: Phương thức accept là "cổng vào" để Visitor tương tác với các đối tượng Person. Mỗi đối tượng Person (BeLa, KIBM, SauPhatSang) tự biết mình nên được "thăm" bằng phương thức visitXXX nào của Visitor, và "chuyển quyền điều khiển" cho Visitor để thực hiện hành động "thăm viếng" đó.

Bước 2: Sửa đổi phương thức visit chính trong class J98.

```
class Visitor :
```

```
    def visit (self , ctx : Person ):
```

```
        return ctx. accept ( self )
```

```
class J98( Visitor ):
```

```
    def visitBeLa (self , ctx: BeLa ):
```

```
        print ("I will give you 3.5 million VND ")
```

```
    def visitKIBM (self , ctx: KIBM ):
```

```
        print ("You oppressed me in the past ")
```

```
    def visitSauPhatSang (self , ctx: SauPhatSang ):
```

```
        print ("I will sing a song for you")
```

Ý nghĩa của hàm visit mới: Hàm visit trong J98 bây giờ trở thành điểm khởi đầu của quá trình "thăm viếng". Khi gọi `j98_object.visit(person_object)`, `j98_object` sẽ ủy quyền cho `person_object` quyết định phương thức `visitXXX` nào sẽ được gọi trên `j98_object`.

Bước 3: "Keep all previously defined visit methods in the J98 class unchanged."

"Keep all previously defined visit methods in the J98 class unchanged.":

Bước này đơn giản là giữ nguyên các hàm `visitBeLa`, `visitKIBM`, `visitSauPhatSang` trong class J98 như chúng đã được định nghĩa ở các slide trước. Các hàm này chứa logic xử lý cụ thể cho từng loại "người".

From simple implementation to visitor pattern (4)

```
1 class Person: pass
2
3 class BeLa(Person):
4     def accept(self, v: Visitor):
5         return v.visitBeLa(self)
6
7 class KIBM(Person):
8     def accept(self, v: Visitor):
9         return v.visitKIBM(self)
10
11 class SauPhatSang(Person):
12     def accept(self, v: Visitor):
13         return v.visitSauPhatSang(self)
```



From simple implementation to visitor pattern (4)



```
1 class Visitor:
2     def visit(self, ctx: Person):
3         return ctx.accept(self)
4
5 class J98(Visitor):
6     def visitBeLa(self, ctx: BeLa):
7         print("I will give you 3.5 million VND")
8
9     def visitKIBM(self, ctx: KIBM):
10        print("You oppressed me in the past")
11
12    def visitSauPhatSang(self, ctx: SauPhatSang):
13        print("I will sing a song for you")
```

Visitor Pattern

cho phép cung cấp nhiều tác vụ đc thêm vào đối tượng đã cho trước mà ko làm thay đổi class đó

Definition

The **Visitor Pattern** allows new operations to be added to existing object structures without modifying their classes. It separates the **data structure** (Elements) from the **operations** (Visitors) performed on them.

Key Components:

- ① Elements (Objects being visited):
 - These are existing classes that need new operations.
 - Each element has an `accept(visitor)` method that **allows a visitor to interact** with it.
- ② Visitors (Operations applied to elements):
 - A visitor contains various `visit(element)` methods for **handling different element types.**
 - New operations can be added without modifying element classes.



Key Components (Các Thành Phần Chính):

Elements (Các Đối tượng được thăm):

"These are existing classes that need new operations.": Elements là các class đã tồn tại và bạn muốn thêm các thao tác mới lên chúng mà không muốn sửa đổi trực tiếp class Elements.

"Each element has an accept(visitor) method that allows a visitor to interact with it.": Mỗi class Element cần có phương thức accept(visitor). Phương thức này đóng vai trò "điểm chấp nhận" Visitor, cho phép Visitor "thăm" và thao tác trên Element. accept thường có logic đơn giản là gọi lại phương thức visitXXX tương ứng của Visitor, truyền self (đối tượng Element) làm tham số.

Visitors (Các Thao Tác được áp dụng lên elements):

"A visitor contains various visit(element) methods for handling different element types.": Visitor là một interface (hoặc abstract class) định nghĩa các phương thức visitXXX(element). Mỗi phương thức visitXXX xử lý một kiểu Element cụ thể (ví dụ visitBeLa(BeLa bela), visitKIBM(KIBM kibm), visitExpr(ExprContext ctx)).

"New operations can be added without modifying element classes.": Ưu điểm quan trọng nhất: Khi bạn muốn thêm một thao tác mới (ví dụ: "J98 đi quyên góp tiền từ mọi người"), bạn chỉ cần tạo một Concrete Visitor mới (ví dụ class J98QuyenGop) mà không cần sửa đổi bất kỳ class Element nào (Person, BeLa, KIBM, SauPhatSang).

grammar BKOOL ;

```
program : expr EOF ;  
expr   : expr ADD term | term ;  
term   : term MUL fact | fact ;  
fact   : LP expr RP | ID | INT ;
```

```
ADD : '+'; MUL: '*';  
ID: [a-zA-Z]+; INT: [0-9]+;  
WS: [ \t\r\n]+ -> skip ;  
LP: '('; RP: ')';
```

Ví dụ về hoạt động: Chúng ta muốn viết một chương trình duyệt qua Parse Tree và in ra biểu thức infix gốc từ cây đó. Ví dụ, nếu Parse Tree là của biểu thức $b + c + d$, chúng ta muốn chương trình in ra đúng chuỗi "b + c + d".

Cách làm BÌNH THƯỜNG (KHÔNG dùng Visitor Pattern):

Nếu không dùng Visitor Pattern, bạn có thể viết một hàm đệ quy để duyệt cây. Hàm này sẽ kiểm tra loại nút hiện tại (ví dụ ExprContext, TermContext, FactContext) và xử lý khác nhau:

CÁCH KHÔNG DÙNG VISITOR PATTERN (ví dụ, KHÔNG OOP)

```
def print_infix(ctx):
    if isinstance(ctx, BKOOLParser.ExprContext):
        if ctx.ADD(): # Nếu là phép cộng
            left_expr = ctx.expr() # Con trái (expr)
            op = ctx.ADD() # Toán tử '+'
            right_term = ctx.term() # Con phải (term)
            return print_infix(left_expr) + " " + op.getText() + " " + print_infix(right_term)
        else: # Trường hợp expr -> term
            return print_infix(ctx.term()) # Chỉ cần xử lý term
    elif isinstance(ctx, BKOOLParser.TermContext):
        if ctx.MUL(): # Nếu là phép nhân
            left_term = ctx.term() # Con trái (term)
            op = ctx.MUL() # Toán tử '*'
            right_fact = ctx.fact() # Con phải (fact)
            return print_infix(left_term) + " " + op.getText() + " " + print_infix(right_fact)
        else: # Trường hợp term -> fact
            return print_infix(ctx.fact()) # Chỉ cần xử lý fact
    elif isinstance(ctx, BKOOLParser.FactContext):
        if ctx.LP(): # Nếu là dấu ngoặc đơn
            expr_in_paren = ctx.expr() # Biểu thức bên trong ngoặc
            return "(" + print_infix(expr_in_paren) + ")"
        elif ctx.ID(): # Nếu là định danh (ID)
            return ctx.ID().getText() # Lấy tên định danh
        elif ctx.INT(): # Nếu là số nguyên (INT)
            return ctx.INT().getText() # Lấy giá trị số nguyên
        return "" # Trường hợp khác (ví dụ ProgramContext, EOF), trả về chuỗi rỗng
```

... (Code để parse input và lấy rootContext) ...

```
infix_expression = print_infix(rootContext)
print(infix_expression)
```

Vấn đề với cách này (tương tự như ví dụ J98 "cách 1"):

Hàm `print_infix` quá lớn và phức tạp: Nó phải xử lý mọi loại nút (`ExprContext`, `TermContext`, `FactContext`, ...) và mọi trường hợp trong từng loại nút (phép cộng, phép nhân, ngoặc đơn, ID, INT).

Khó mở rộng: Nếu bạn muốn thêm một hoạt động mới (ví dụ, kiểm tra kiểu dữ liệu, tính giá trị biểu thức), bạn sẽ phải sửa lại hàm `print_infix` hoặc viết thêm các hàm tương tự, làm code càng rối rắm.

Không OOP: Code không được tổ chức theo hướng đối tượng, logic xử lý nút bị "trộn lẫn" vào hàm `print_infix`.

Cách dùng VISITOR PATTERN (OOP và dễ mở rộng hơn):

Chúng ta sẽ tạo một `Visitor` class (ví dụ `InfixPrinter`) để thực hiện việc in biểu thức infix.

CÁCH DÙNG VISITOR PATTERN

```
from BKOOLVisitor import BKOOLVisitor # Import Visitor base class do ANTLR4 sinh ra
from BKOOLParser import BKOOLParser
```

```
class InfixPrinter(BKOOLVisitor): # Tạo class InfixPrinter kế thừa BKOOLVisitor
```

```
def visitExpr(self, ctx: BKOOLParser.ExprContext):
    if ctx.ADD(): # Nếu là phép cộng
        left_expr = ctx.expr() # Con trái (expr)
        op = ctx.ADD() # Toán tử '+'
        right_term = ctx.term() # Con phải (term)
        # Gọi visit trên các con và kết hợp kết quả
        return self.visit(left_expr) + " " + op.getText() + " " + self.visit(right_term)
    else: # Trường hợp expr -> term
        return self.visit(ctx.term()) # Gọi visit trên term
```

```
def visitTerm(self, ctx: BKOOLParser.TermContext):
    if ctx.MUL(): # Nếu là phép nhân
        left_term = ctx.term() # Con trái (term)
        op = ctx.MUL() # Toán tử '*'
        right_fact = ctx.fact() # Con phải (fact)
        # Gọi visit trên các con và kết hợp kết quả
        return self.visit(left_term) + " " + op.getText() + " " + self.visit(right_fact)
    else: # Trường hợp term -> fact
        return self.visit(ctx.fact()) # Gọi visit trên fact
```

```
def visitFact(self, ctx: BKOOLParser.FactContext):
    if ctx.LP(): # Nếu là dấu ngoặc đơn
        expr_in_paren = ctx.expr() # Biểu thức bên trong ngoặc
        # Gọi visit trên expr và thêm dấu ngoặc
        return "(" + self.visit(expr_in_paren) + ")"
    elif ctx.ID(): # Nếu là định danh (ID)
        return ctx.ID().getText() # Lấy tên định danh
    elif ctx.INT(): # Nếu là số nguyên (INT)
        return ctx.INT().getText() # Lấy giá trị số nguyên
    return "" # Trường hợp khác, trả về chuỗi rỗng
```

```
# ... (Code để parse input và lấy rootContext) ...
```

```
infix_visitor = InfixPrinter() # Tạo đối tượng InfixPrinter (Visitor)
infix_expression = infix_visitor.visit(rootContext) # Gọi visit trên rootContext để bắt đầu duyệt cây
print(infix_expression)
```

Giải thích code Visitor Pattern:

`class InfixPrinter(BKOOOLVisitor)::` Chúng ta tạo một class `InfixPrinter` kế thừa từ `BKOOOLVisitor`. `BKOOOLVisitor` là `Visitor` interface cơ sở do ANTLR4 tự động sinh ra từ grammar `BKOOOL.g4`. Class `InfixPrinter` sẽ là Concrete Visitor của chúng ta.

`def visitExpr(self, ctx: BKOOOLParser.ExprContext):`, `def visitTerm(self, ctx: BKOOOLParser.TermContext):`, `def visitFact(self, ctx: BKOOOLParser.FactContext):`:: Chúng ta override (ghi đè) các phương thức `visitExpr`, `visitTerm`, `visitFact` trong `InfixPrinter`. Các phương thức này tương ứng với các quy tắc parser `expr`, `term`, `fact` trong grammar. Mỗi phương thức này sẽ xử lý một loại nút cụ thể trong Parse Tree.

Bên trong mỗi phương thức `visitXXX`:

Chúng ta kiểm tra xem nút hiện tại (`ctx`) thuộc trường hợp nào của quy tắc tương ứng (ví dụ, trong `visitExpr`, kiểm tra xem có phải phép cộng `ctx.ADD()` hay không).

Chúng ta truy cập các nút con bằng các phương thức như `ctx.expr()`, `ctx.term()`, `ctx.fact()`, `ctx.ADD()`, `ctx.ID()`, `ctx.INT()`, ... (như đã giải thích ở slide "Common Ways to Access Children").

Quan trọng nhất: Chúng ta gọi lại `self.visit(child_ctx)` đệ quy trên các nút con. `self.visit(...)` chính là cách kích hoạt cơ chế duyệt cây của Visitor Pattern. Khi bạn gọi `self.visit(child_ctx)`, Visitor Pattern sẽ tự động xác định kiểu của `child_ctx` (ví dụ `TermContext`) và gọi đúng phương thức `visitTerm(child_ctx)` trong class `InfixPrinter`. Đây là cách Visitor Pattern "điều hướng" việc xử lý đến đúng phương thức dựa trên kiểu nút.

Chúng ta xây dựng chuỗi infix bằng cách kết hợp kết quả trả về từ việc gọi `self.visit` trên các con và các token (ví dụ `op.getText()`).

`infix_visitor = InfixPrinter()`, `infix_expression = infix_visitor.visit(rootContext)`:

Chúng ta tạo một đối tượng `infix_visitor` thuộc class `InfixPrinter` (Concrete Visitor).

Chúng ta gọi phương thức `visit` một lần duy nhất trên nút gốc `rootContext` của Parse Tree, bắt đầu quá trình duyệt cây. Cơ chế Visitor Pattern sẽ tự động điều khiển việc duyệt cây từ nút gốc, gọi các phương thức `visitXXX` tương ứng trên `infix_visitor` khi duyệt đến từng loại nút.

Ưu điểm của cách dùng Visitor Pattern trong ví dụ này:

Code có cấu trúc OOP: Logic in infix được tách biệt hoàn toàn khỏi grammar và cấu trúc Parse Tree, đóng gói trong class `InfixPrinter`.

Dễ mở rộng: Nếu bạn muốn thêm một hoạt động mới (ví dụ, tính giá trị biểu thức), bạn chỉ cần tạo một Visitor class mới (ví dụ `ExpressionEvaluator` kế thừa `BKOOOLVisitor`) và override các phương thức `visitXXX` theo logic tính giá trị, mà không cần sửa đổi class `InfixPrinter` hay grammar.

Code dễ bảo trì: Mỗi phương thức `visitXXX` nhỏ gọn, chỉ xử lý một loại nút cụ thể, dễ đọc và dễ sửa lỗi.



Understanding
Abstract Syntax Trees
(AST)

Traversing parse tree

Visitor pattern

Using Visitor in
ANTLR4

USING VISITOR IN ANTLR4

Visitor class generated by ANTLR4

In 7 files generated in the `target` directory (for example, grammar BKOOL), class `BKOOLVisitor` is generated in file **`BKOOLVisitor.py`**:

- Each parser rule (a parser rule) will have a corresponding method to process it, named using the form `visit<ParserRuleName>`, where the first letter of the rule name is capitalized.

Example:

- `program` will have the method `visitProgram`.
 - `for_stmt` will have the method `visitFor_stmt`.
- To generate the AST, we will inherit from this class and implement each method specifically.

Để tạo ra AST, chúng ta sẽ kế thừa từ class này và triển khai từng phương thức một cách cụ thể.



Tips for Writing AST Generation

Bắt đầu từ "dưới lên": Nên bắt đầu triển khai các phương thức visitXXX cho các quy tắc parser ở mức "thấp" hơn trong grammar tức là những quy tắc thường chứa nhiều token (terminal symbols) và ít hoặc không chứa các quy tắc non-terminal khác.

- 1 Implement methods for parser rules **from the simplest** (rules containing the most terminal symbols) to the **most complex**, typically starting with the lower-level rules among those already implemented.
Quy tắc parser có nhiều lựa chọn: Nếu một quy tắc parser có nhiều lựa chọn (alternatives - ví dụ dùng | trong BNF), thì phương thức visitXXX tương ứng cũng cần xử lý tất cả các trường hợp lựa chọn đó.
- 2 A parser rule has as many cases as the **visit** method must handle.
ký tự ko kết thúc
- 3 Whenever a non-terminal signal is encountered, **call visit**, but the returned result must be understood for further processing. If unsure how to handle it, use the + operator to join the list or distribute it into a separate list.
Focus vào con trực tiếp: Trong mỗi phương thức visitXXX, bạn thường chỉ cần xử lý trực tiếp các nút con của nút hiện tại (ctx). Bạn không cần phải lo lắng về việc duyệt sâu hơn vào các cháu (con của con).
- 4 Only the child nodes are considered, while the child nodes of those children are ignored.
- 5 During tree generation, it is necessary to continuously compare with AST classes. If enough data is available to create an object, it should be created and returned immediately.

These rules are almost entirely consistent with grammars written in BNF format.



Tạo AST object "vừa đủ": Khi bạn đã thu thập đủ thông tin cần thiết từ nút Parse Tree hiện tại và các nút con (thông qua các lệnh gọi self.visit), bạn nên tạo một đối tượng AST (instance của một AST class) và trả về đối tượng AST đó từ phương thức visitXXX.

"Trả về ngay lập tức": "Immediately" ở đây có nghĩa là, sau khi tạo AST object, bạn return nó ngay, kết thúc việc xử lý nút hiện tại và trả kết quả lên nút cha (nút gọi visitXXX này). Đây là cách xây dựng AST từ dưới lên (bottom-up): nút lá AST được tạo trước, rồi nút cha AST được tạo sau dựa trên các nút con AST đã được tạo.

THANK YOU.

ASTGen

MEng. Tran Ngoc
Bao Duy



Understanding
Abstract Syntax Trees
(AST)

Traversing parse tree

Visitor pattern

Using Visitor in
ANTLR4