Syntax Analysis

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

Terminal (Kí hiệu kết thúc / Từ vựng):

Định nghĩa ngắn gọn: Đây là những kí hiệu cơ bản, không thể chia nhỏ hơn của ngôn ngữ. Chúng là những thứ mà bạn thực sự viết trong code.
Ví dụ: Trong ngôn ngữ lập trình, terminal có thể là:
Từ khóa: if, else, while, int, function, return, ...
Toán tử: +, -, *, /, =, ==, <, >, ...
Hằng số: 123, 3.14, "hello", true, false, ...
Dấu câu: ;, ,, (,), {, }, [,], ...
Tên biến/ Định danh (identifier): x, count, myVariable, calculateSum, ... (với điều kiện chúng đã được xác định là tên biến, chứ không phải là một non-terminal)
Liên hệ: Hãy tưởng tượng terminal giống như từ vựng trong tiếng Việt. Chúng là những từ cơ bản mà bạn dùng để tạo thành câu.

Non-terminal (Kí hiệu chưa kết thúc / Loại cú pháp):

Định nghĩa ngắn gọn: Đây là những kí hiệu đại diện cho các cấu trúc cú pháp phức tạp hơn trong ngôn ngữ. Chúng không trực tiếp xuất hiện trong code cuối cùng, mà là khái niệm trung gian để định nghĩa ngữ pháp. Non-terminal có thể được phân tích tiếp thành các terminal và non-terminal khác theo các quy tắc.
Ví dụ: Trong ngôn ngữ lập trình, non-terminal có thể là:
<statement> (câu lệnh)
<expression> (biểu thức)
<if-statement> (câu lệnh if)
<while-loop> (vòng lặp while)
<function-definition> (định nghĩa hàm)
<variable-declaration> (khai báo biến)
<data-type> (kiểu dữ liệu)
Liên hệ: Non-terminal giống như loại từ vựng (danh từ, động từ, tính từ,...) hoặc cụm từ, mệnh đề, câu trong tiếng Việt. Chúng là các đơn vị ngữ pháp lớn hơn được xây dựng từ từ vựng, và lại có thể tạo nên các đơn vị ngữ pháp lớn hơn nữa.

Production Rule (Quy tắc sản xuất / Quy tắc ngữ pháp):

Định nghĩa ngắn gọn: Đây là một quy tắc mô tả cách một non-terminal có thể được tạo thành từ các terminal và/hoặc non-terminal khác. Quy tắc sản xuất thường được viết dưới dạng: non-terminal → tổ hợp của terminals và/hoặc non-terminals.

Ví dụ:

<if-statement> ::= if (<expression>) { <statement> } (Một câu lệnh if được tạo thành từ từ khóa if, dấu ngoặc đơn (), một <expression>, dấu ngoặc nhọn {}, và một <statement>)

<expression> ::= <term> | <expression> + <term> (Một biểu thức có thể là một <term> HOẶC một <expression> cộng với một <term>)

<term> ::= <factor> | <term> * <factor>

<factor> ::= <number> | (<expression>) | <variable>

Liên hệ: Quy tắc sản xuất giống như quy tắc ngữ pháp trong tiếng Việt (ví dụ: "Câu = Chủ ngữ + Vị ngữ", "Cụm danh từ = (Tính từ) + Danh từ"). Chúng chỉ ra cách bạn kết hợp các thành phần nhỏ hơn để tạo thành các thành phần lớn hơn trong ngôn ngữ.

Grammar (Ngữ pháp):

Định nghĩa ngắn gọn: Một tập hợp các quy tắc sản xuất định nghĩa cú pháp của một ngôn ngữ. Grammar xác định tất cả các chuỗi (code) hợp lệ trong ngôn ngữ đó.

Ví dụ: Một grammar cho một ngôn ngữ số học đơn giản có thể bao gồm các quy tắc sản xuất cho <expression>, <term>, <factor>, <number>, <operator>, ...

Liên hệ: Grammar giống như toàn bộ hệ thống ngữ pháp của tiếng Việt, bao gồm tất cả các quy tắc về từ vựng, cấu trúc câu, dấu câu, ... Nó định nghĩa cách viết đúng tiếng Việt.

BNF (Backus-Naur Form) & EBNF (Extended Backus-Naur Form):

Định nghĩa ngắn gọn: Đây là các kí hiệu (notations) chuẩn để viết grammar phi ngữ cảnh (context-free grammar).

BNF là kí hiệu gốc, dùng đệ quy để thể hiện lặp lại và tùy chọn.

EBNF là mở rộng của BNF, thêm các toán tử RE (*, +, ?, |, ()) để viết grammar ngắn gọn và dễ đọc hơn. Cả BNF và EBNF đều mô tả cùng loại ngôn ngữ (CFLs).

Ví dụ: Bạn có thể dùng BNF hoặc EBNF để viết grammar cho ngôn ngữ lập trình C, Java, Python, ... hoặc ngôn ngữ XML, JSON, ...

Liên hệ: BNF và EBNF giống như bộ công cụ hoặc cách viết chuẩn để mô tả ngữ pháp của một ngôn ngữ một cách chính xác và rõ ràng. EBNF giống như phiên bản "nâng cấp" của BNF, có thêm nhiều công cụ tiện lợi hơn.

Regular Expressions (RE) / Biểu thức chính quy:

Định nghĩa ngắn gọn: Đây là một ngôn ngữ mô tả mẫu chuỗi mạnh mẽ, thường được dùng để tìm kiếm, so khớp, và thao tác chuỗi. RE rất tốt cho việc mô tả các mẫu tuyến tính (ví dụ: định dạng email, số điện thoại), nhưng yếu trong việc mô tả cấu trúc lồng nhau.

Ví dụ: Các mẫu RE:

[a-zA-Z]+ (một hoặc nhiều chữ cái)

\d* (0 hoặc nhiều chữ số)

(if|while|for) (từ khóa if hoặc while hoặc for)

Liên hệ: RE giống như công cụ tìm kiếm mẫu nâng cao cho văn bản. Nó giúp bạn tìm ra các chuỗi kí tự có dạng nhất định.

Context-Free Grammar (CFG) / Grammar phi ngữ cảnh:

Định nghĩa ngắn gọn: Đây là một loại grammar mạnh hơn Regular Grammar (mà RE mô tả). CFG có khả năng mô tả các cấu trúc lồng nhau và đệ quy (ví dụ: biểu thức ngoặc đơn, khối lệnh, cấu trúc HTML). BNF và EBNF là các kí hiệu để viết CFGs.

Ví dụ: Grammar cho ngôn ngữ lập trình thường là CFG vì chúng cần mô tả các cấu trúc lồng nhau phức tạp.

Liên hệ: CFG giống như hệ thống ngữ pháp đầy đủ của một ngôn ngữ phức tạp (như ngôn ngữ lập trình hoặc ngôn ngữ tự nhiên). Nó đủ mạnh để mô tả các quy tắc phức tạp về cấu trúc và thứ tự.

Context-Free Language (CFL) / Ngôn ngữ phi ngữ cảnh:

Định nghĩa ngắn gọn: Đây là tập hợp tất cả các chuỗi (code) có thể được tạo ra (dẫn xuất) bởi một Context-Free Grammar (CFG). Nói cách khác, CFL là ngôn ngữ được định nghĩa bởi một CFG. BNF và EBNF có thể mô tả CFLs.

Ví dụ: Tập hợp tất cả các chương trình C hợp lệ, tập hợp tất cả các tài liệu XML hợp lệ, đều là các ví dụ về CFLs (hoặc gần đúng là CFLs).

Liên hệ: CFL giống như tập hợp tất cả các câu đúng ngữ pháp trong tiếng Việt (nếu chúng ta coi ngữ pháp tiếng Việt là một CFG gần đúng).

Bảng tóm tắt nhanh:

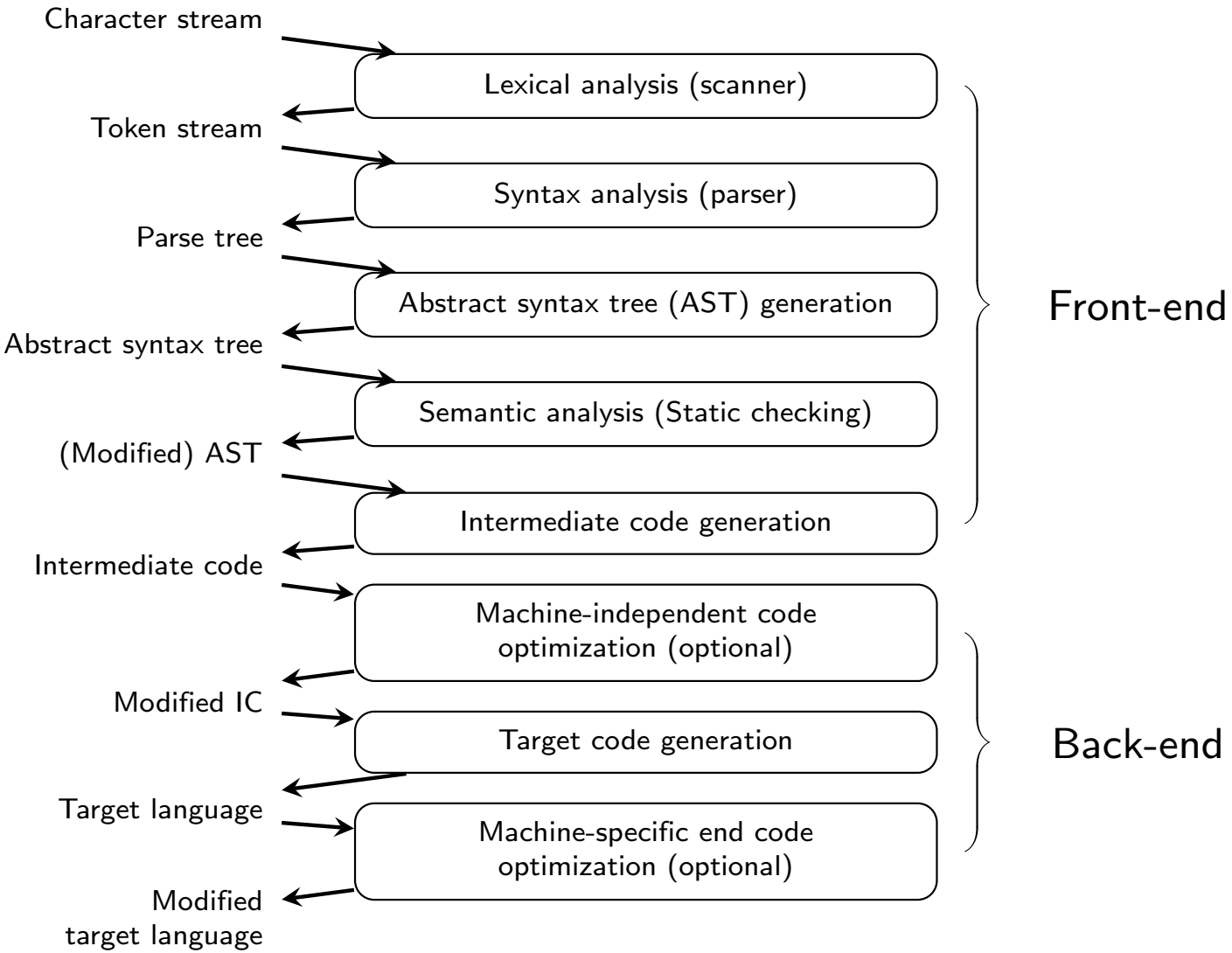
Thuật ngữ	Định nghĩa ngắn gọn	Liên hệ ví dụ	Mạnh hơn/Yếu hơn	Dùng để làm gì?
Terminal	Kí hiệu cơ bản, không chia nhỏ được	Từ khóa, toán tử, hằng số, ...	Cơ bản nhất	Thành phần cơ bản của code
Non-terminal	Kí hiệu đại diện cấu trúc cú pháp, có thể phân tích tiếp	`<statement>`, `<expression>`, `<if-statement>`, ...	Lớn hơn Terminal	Định nghĩa cấu trúc cú pháp
Production Rule	Quy tắc tạo non-terminal từ terminals/non-terminals	`<if-statement> ::= ...`, `<expression> ::= ...`	Quy tắc	Định nghĩa ngữ pháp
Grammar	Tập hợp các quy tắc sản xuất	Grammar cho C, Java, XML, ...	Hệ thống	Định nghĩa cú pháp toàn bộ ngôn ngữ
BNF	Kí hiệu viết CFG, dùng đệ quy	Ví dụ BNF cho biểu thức số học	Kí hiệu	Mô tả CFG, định nghĩa cú pháp ngôn ngữ
EBNF	Mở rộng của BNF, thêm RE operators, ngắn gọn hơn BNF	Ví dụ EBNF cho danh sách số	Kí hiệu	Mô tả CFG (ngắn gọn hơn), định nghĩa cú pháp ngôn ngữ
RE	Ngôn ngữ mô tả mẫu chuỗi, tốt cho mẫu tuyến tính	`[a-zA-Z]+`, `\d*`, `(if while)`, ...	Yếu hơn CFG	
CFG	Loại Grammar mạnh, mô tả cấu trúc lồng nhau, đệ quy	Grammar cho ngôn ngữ lập trình phức tạp	Mạnh hơn RE	Định nghĩa cú pháp ngôn ngữ lập trình, XML, ...
CFL	Ngôn ngữ được định nghĩa bởi CFG	Tập hợp code C hợp lệ, XML hợp lệ, ...	Ngôn ngữ	Ngôn ngữ được mô tả bởi CFG

Overview

INTRODUCTION TO SYNTAX ANALYSIS

MEng. Tran Ngoc Bao Duy

An overview of compilation



MEng. Tran Ngoc Bao Duy

Roles of Syntax Analysis

Definition

Syntax analysis, also known as parsing, is the second phase of a compiler, following lexical analysis. Its primary role is to check whether the sequence of tokens generated by the lexical analyzer forms a valid syntactic structure according to the rules of the programming language's grammar.

1. **Syntax Validation**: kiểm tra có ngữ pháp nào hình thành nên chuỗi đó ko
   - Ensures that the program's structure follows the language's grammar rules.
   - Detects syntax errors and reports them to the developer with clear error messages.

dựng parse tree thể hiện quá trình sinh ra chuỗi token đó

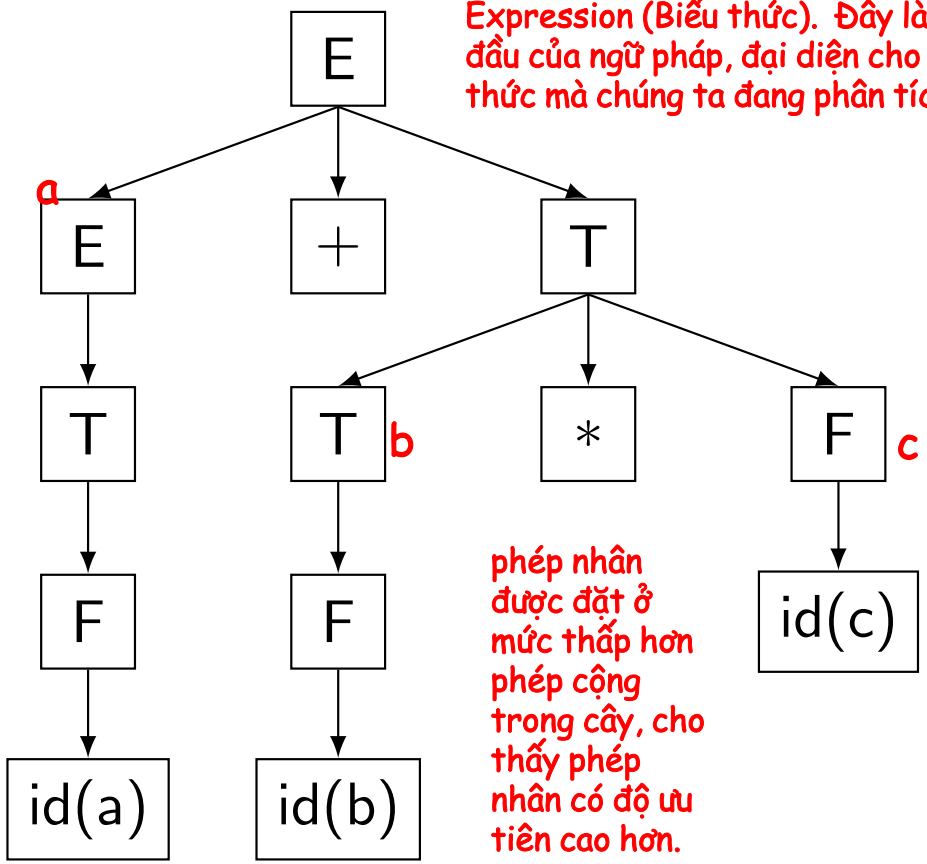
2. **Parse Tree Generation**: Constructs a parse tree (or abstract syntax tree - AST) representing the hierarchical structure of the program based on grammar rules.

Nếu chương trình không có lỗi cú pháp (vượt qua giai đoạn Syntax Validation), Parser sẽ tiến hành tạo ra một cấu trúc dữ liệu gọi là parse tree (cây phân tích cú pháp) hoặc Abstract Syntax Tree (cây cú pháp trừu tượng - AST).v

MEng. Tran Ngoc Bao Duy

Syntax Analysis: Example

- Input: $a + b * c$
- Grammar rules (Simplified Context-Free Grammar)
- Output (as parse tree):



Chữ E ở trên cùng thường là viết tắt của Expression (Biểu thức). Đây là ký hiệu bắt đầu của ngữ pháp, đại diện cho toàn bộ biểu thức mà chúng ta đang phân tích.

phép nhân được đặt ở mức thấp hơn phép cộng trong cây, cho thấy phép nhân có độ ưu tiên cao hơn.

CONTEXT-FREE GRAMMAR

MEng. Tran Ngoc Bao Duy

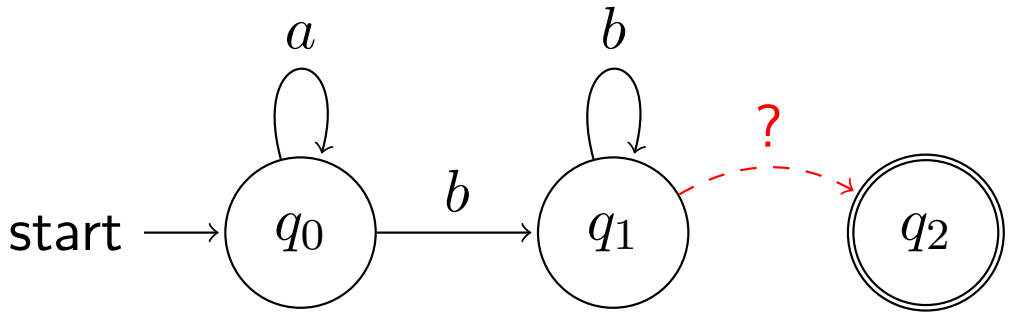
The Limits of Regular Expressions

Biểu thức Chính quy rất hữu ích để tìm kiếm và xác định các mẫu văn bản đơn giản. Ví dụ, bạn có thể dùng REs để tìm tất cả các địa chỉ email trong một văn bản, hoặc kiểm tra xem một chuỗi có đúng định dạng số điện thoại hay không.

Regular expressions (REs) are powerful for pattern matching. But can they express certain symmetric patterns?

$$L = \{a^n b^n \mid n > 0\}$$

Using a finite automaton (as back-end of regex):



Có những loại mẫu mà REs không thể mô tả được.

"L = {aⁿbⁿ | n > 0}" - Đây là ví dụ về một ngôn ngữ (L). Ký hiệu này định nghĩa một tập hợp các chuỗi:

aⁿbⁿ: Có nghĩa là chuỗi bắt đầu bằng n ký tự 'a' (aⁿ), sau đó là n ký tự 'b' (bⁿ).
n > 0: Có nghĩa là số lượng 'a' và 'b' phải lớn hơn 0 (ít nhất một chữ 'a' và một chữ 'b').

MEng. Tran Ngoc Bao Duy

Dấu chấm hỏi màu đỏ và ý nghĩa: Dấu chấm hỏi ở đây thể hiện sự bất lực của Finite Automaton. FA không có bộ nhớ để đếm số lượng 'a' đã đọc và sau đó so sánh với số lượng 'b' để đảm bảo chúng bằng nhau. FA chỉ có thể dựa vào trạng thái hiện tại và ký tự đầu vào hiện tại để quyết định chuyển trạng thái. Nó không thể "nhớ" thông tin từ quá khứ (ví dụ: đã đọc bao nhiêu 'a' rồi).

iểu thức Chính quy (REs) về mặt lý thuyết tương đương với máy tự động hữu hạn (finite automaton - FA). Điều này có nghĩa là bất kỳ cái gì có thể được biểu diễn bằng REs thì cũng có thể được nhận diện bởi một FA, và ngược lại. Hình ảnh bên dưới minh họa cố gắng xây dựng một FA để nhận diện ngôn ngữ L = {aⁿbⁿ | n > 0}.

Giải thích sơ đồ Finite Automaton:

start -> q₀: q₀ là trạng thái bắt đầu.
Vòng lặp 'a' trên q₀: Khi đọc ký tự 'a' ở trạng thái q₀, máy sẽ vẫn ở lại trạng thái q₀. Vòng lặp này cố gắng xử lý phần "aⁿ".
q₀ -> q₁ (đọc 'b'): Khi đọc ký tự 'b' lần đầu tiên ở trạng thái q₀, máy sẽ chuyển sang trạng thái q₁. Điều này đánh dấu sự kết thúc của phần 'a' và bắt đầu phần 'b'.
Vòng lặp 'b' trên q₁: Khi đọc ký tự 'b' ở trạng thái q₁, máy sẽ vẫn ở lại trạng thái q₁. Vòng lặp này cố gắng xử lý phần "bⁿ".
q₁ -> q₂ (đọc '?'): Đây là phần có dấu chấm hỏi màu đỏ. Sau khi đọc một hoặc nhiều ký tự 'b' ở trạng thái q₁, chúng ta muốn kết thúc và chấp nhận chuỗi. Trạng thái q₂ (vòng tròn kép) là trạng thái chấp nhận. Tuy nhiên, vấn đề là không có ký tự cụ thể nào để chuyển từ q₁ sang q₂. Chúng ta muốn chuyển khi đã đọc đủ số lượng 'b' tương ứng với số lượng 'a' đã đọc trước đó.
Dấu chấm hỏi màu đỏ và ý nghĩa: Dấu chấm hỏi ở đây thể hiện sự bất lực của Finite Automaton. FA không có bộ nhớ để đếm số lượng 'a' đã đọc và sau đó so sánh với số lượng 'b' để đảm bảo chúng bằng nhau. FA chỉ có thể dựa vào trạng thái hiện tại và ký tự đầu vào hiện tại để quyết định chuyển trạng thái. Nó không thể "nhớ" thông tin từ quá khứ (ví dụ: đã đọc bao nhiêu 'a' rồi).

Kết luận về giới hạn của Regular Expressions (và Finite Automaton):

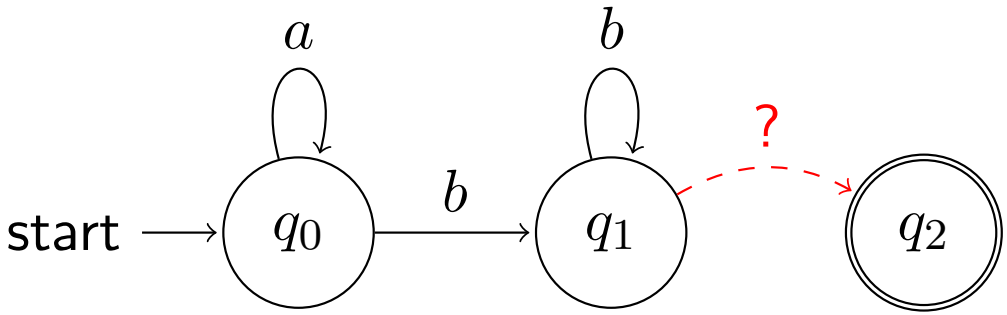
Hình ảnh này minh họa rằng Biểu thức Chính quy (và Máy tự động hữu hạn) không đủ mạnh để nhận diện ngôn ngữ L = {aⁿbⁿ | n > 0}. Lý do là vì ngôn ngữ này đòi hỏi khả năng đếm và so sánh, là những thứ mà FA (và do đó, REs) không thể thực hiện được.

The Limits of Regular Expressions

Regular expressions (REs) are powerful for pattern matching. But can they express certain symmetric patterns?

$$L = \{a^n b^n \mid n > 0\}$$

Using a finite automaton (as back-end of regex):



Limitation: The finite automaton cannot count the number of as to ensure they match the number of bs.

MEng. Tran Ngoc Bao Duy

Symmetry in Programming

đối xứng

Symmetry in programming refers to structures where one part mirrors or matches another, ensuring balance and correctness.

- Balanced Parentheses: `((...))`
- Control Structures: `if ... then ... else` or `repeat ... until`
- HTML/XML Tags: `<div><p>Text</p></div>`

Cấu trúc if thường có hai nhánh chính: nhánh then (hoặc if) thực hiện khi điều kiện đúng, và nhánh else thực hiện khi điều kiện sai. Hai nhánh này tạo thành một cặp, đảm bảo chương trình sẽ luôn đi theo một trong hai hướng, không bị "mắc kẹt" hay thiếu hướng đi.

repeat ... until (Lặp lại ... cho đến khi):
Đối xứng: Cấu trúc vòng lặp repeat ... until có điểm bắt đầu repeat và điểm kết thúc until <điều kiện>. Hai từ khóa này tạo thành một cặp, giới hạn phạm vi của khối lệnh được lặp lại.
Tính cân bằng: Sự đối xứng giúp xác định rõ ràng phần code nào sẽ được lặp lại và điều kiện dừng vòng lặp, tránh vòng lặp vô hạn hoặc lặp không đúng số lần.

Các cấu trúc điều khiển khác cũng có tính đối xứng: Ví dụ, for (initialization; condition; increment) { ... } có các phần khởi tạo, điều kiện, và bước nhảy, tạo thành một cấu trúc có tổ chức và dễ kiểm soát.

MEng. Tran Ngoc Bao Duy

Symmetry in Programming

Symmetry in programming refers to structures where one part mirrors or matches another, ensuring balance and correctness.

- Balanced Parentheses: `((...))`
- Control Structures:

  `if ... then ... else` or `repeat ... until`
- HTML/XML Tags: `<div><p>Text</p></div>`

Regular expressions cannot describe this kind of structure but they could be:

1. A means of describing this kind of language.
2. A method to detect if a sequence of tokens is valid or invalid with respect to this kind of language.

MEng. Tran Ngoc Bao Duy

(Biểu thức chính quy không thể mô tả loại cấu trúc này): Câu này khẳng định lại kết luận quan trọng mà chúng ta đã rút ra từ hình ảnh về máy tự động hữu hạn. "Loại cấu trúc này" ở đây chính là tính đối xứng và khả năng đếm và so sánh số lượng như trong ngôn ngữ L = {aⁿbⁿ | n > 0}. REs (và Finite Automaton) không đủ mạnh để mô tả những ngôn ngữ như vậy. Chúng bị giới hạn ở việc mô tả các mẫu tuyến tính, không có khả năng "ghi nhớ" và "đối sánh" số lượng giữa các phần của chuỗi.

Context-free grammar

CFG là một công cụ toán học được xây dựng một cách chính xác và chặt chẽ. "Hệ thống hình thức" có nghĩa là nó dựa trên các quy tắc và ký hiệu được xác định rõ ràng, không mơ hồ.

A **context-free grammar (CFG)** is a formal system used to define programming languages, natural languages, and structures in computation. CFG generates strings by applying **production rules** that replace non-terminal symbols with sequences of symbols.

Components

1. **A set of terminals** $\mathbf{T}$: Basic symbols (e.g., A, B) used to represent strings.
2. **A set of non-terminals** $\mathbf{N}$: Abstract symbols (e.g., S, X) used in rules.
3. **A start symbol** $\mathbf{S \in N}$**:** A special non-terminal where derivation starts.
4. **A set of production** $\mathbf{P}$: Rules for replacing non-terminals with sequences of terminals/non-terminals.
   A production $\mathbf{p} \in \mathbf{P}$ is in the form: $X \to \alpha$ where $X \in \mathbf{N}$ and α is a sequence of symbols in $\mathbf{T}$ and/or $\mathbf{N}$.

MEng. Tran Ngoc Bao Duy

Slide chia CFG thành 4 thành phần chính, được đánh số từ ① đến ④. Đây là những "mảnh ghép" cơ bản để xây dựng một CFG:
① A set of terminals T: Basic symbols (e.g., A, B) used to represent strings. (① Một tập hợp các ký hiệu kết thúc T: Các ký hiệu cơ bản (ví dụ: A, B) được sử dụng để biểu diễn các chuỗi.)
Terminals (Ký hiệu kết thúc): Đây là các ký hiệu cơ bản, không thể chia nhỏ hơn, và là thành phần cuối cùng trong các chuỗi mà CFG tạo ra. Chúng giống như "từ vựng" của ngôn ngữ.
Ví dụ (e.g., A, B): Ví dụ đơn giản nhất là các chữ cái in hoa A, B,... Trong ngữ cảnh ngôn ngữ lập trình, terminals thường là các tokens được tạo ra bởi Lexical Analyzer (phân tích từ vựng). Ví dụ:
Từ khóa: if, else, while, int, return,...
Toán tử: +, -, *, /, =, ==, <, >,...
Định danh (Identifiers): Tên biến, tên hàm,... (ví dụ: x, count, calculateTotal,...)
Hằng số (Constants): Số, chuỗi ký tự,... (ví dụ: 10, 3.14, "Hello",...)
Dấu câu/ký hiệu đặc biệt: ;, ,, (,), {, }, [,],...
② A set of non-terminals N: Abstract symbols (e.g., S, X) used in rules. (② Một tập hợp các ký hiệu không kết thúc N: Các ký hiệu trừu tượng (ví dụ: S, X) được sử dụng trong các quy tắc.)
Non-terminals (Ký hiệu không kết thúc): Đây là các ký hiệu trừu tượng, đại diện cho các cấu trúc ngữ pháp. Chúng không xuất hiện trong chuỗi kết quả cuối cùng, mà chỉ được sử dụng trong các luật sinh để xây dựng cấu trúc. Chúng giống như "danh mục ngữ pháp" hoặc "khái niệm ngữ pháp".
Ví dụ (e.g., S, X): Ví dụ đơn giản là các chữ cái in hoa như S, X, Y, E, T, F,... (thường dùng các chữ cái đầu của các khái niệm ngữ pháp). Trong ngữ cảnh ngôn ngữ lập trình, non-terminals thường đại diện cho các cấu trúc ngữ pháp như:
Expression (Biểu thức)
Statement (Câu lệnh)
Block (Khối lệnh)
Declaration (Khai báo)
Type (Kiểu dữ liệu)
FunctionCall (Gọi hàm)
Condition (Điều kiện)
... và nhiều cấu trúc ngữ pháp khác.
A start symbol S∈ N: A special non-terminal where derivation starts. (Một ký hiệu bắt đầu S∈ N: Một ký hiệu không kết thúc đặc biệt nơi quá trình dẫn xuất bắt đầu.)
Start symbol (Ký hiệu bắt đầu): Đây là một non-terminal đặc biệt được chọn làm điểm khởi đầu cho quá trình derivation (dẫn xuất), tức là quá trình tạo ra chuỗi từ ngữ pháp. Thường ký hiệu bắt đầu được đặt tên là S (viết tắt của Start hoặc Sentence).
Vai trò: Quá trình dẫn xuất luôn bắt đầu từ ký hiệu bắt đầu. Từ ký hiệu bắt đầu, chúng ta sẽ áp dụng các luật sinh để dần dần thay thế các non-terminal cho đến khi chỉ còn lại các terminal. Chuỗi các terminal cuối cùng chính là chuỗi được sinh ra bởi ngữ pháp.
④ A set of production P: Rules for replacing non-terminals with sequences of terminals/non-terminals. (④ Một tập hợp các luật sinh P: Các quy tắc để thay thế các ký hiệu không kết thúc bằng các chuỗi ký hiệu kết thúc/không kết thúc.)
Production rules (Luật sinh): Đây là trái tim của CFG. Luật sinh định nghĩa cách các non-terminal có thể được thay thế bằng các chuỗi khác. Mỗi luật sinh có dạng:
X → α
X: Là một non-terminal (thuộc tập N), nằm ở vế trái của luật.
→: Ký hiệu mũi tên, có nghĩa là "có thể được thay thế bởi" hoặc "sinh ra".
α: Là một chuỗi ký hiệu (thuộc (T ∪ N)[*]), nằm ở vế phải của luật. Chuỗi này có thể chứa:
Terminals (T): Các ký hiệu kết thúc.
Non-terminals (N): Các ký hiệu không kết thúc khác.
ε (epsilon): Chuỗi rỗng (trong một số trường hợp, nếu ngữ pháp cho phép).
Ví dụ về luật sinh:
E → E + T (Biểu thức có thể là Biểu thức cộng với Thành phần)
E → T (Biểu thức có thể chỉ là Thành phần)
T → T * F (Thành phần có thể là Thành phần nhân với Nhân tố)
T → F (Thành phần có thể chỉ là Nhân tố)
F → id (Nhân tố có thể là định danh - identifier)
F → num (Nhân tố có thể là số - number)
F → (E) (Nhân tố có thể là biểu thức đặt trong ngoặc đơn)
S → Statement ; S (Chương trình có thể là Câu lệnh theo sau bởi dấu chấm phẩy và Chương trình khác)
S → Statement (Chương trình có thể chỉ là một Câu lệnh)
Statement → if Condition then Block else Block (Câu lệnh có thể là cấu trúc if-then-else)
Block → { StatementList } (Khối lệnh có thể là dấu ngoặc nhọn bao quanh danh sách câu lệnh)

CFG được sử dụng để định nghĩa ngôn ngữ lập trình, ngôn ngữ tự nhiên và cấu trúc trong tính toán.): Câu này chỉ ra ứng dụng rộng rãi của CFG. Mặc dù chúng ta đang tập trung vào ngôn ngữ lập trình, CFG còn được dùng trong nhiều lĩnh vực khác:

(CFG tạo ra các chuỗi bằng cách áp dụng các luật sinh mà thay thế các ký hiệu không kết thúc bằng các chuỗi ký hiệu.) - Đây là cơ chế hoạt động chính của CFG. CFG tạo ra (sinh ra) các chuỗi (ví dụ, các chương trình hợp lệ) thông qua việc sử dụng production rules (luật sinh). Các luật này hoạt động bằng cách thay thế các non-terminal symbols (ký hiệu không kết thúc) bằng các sequences of symbols (chuỗi ký hiệu) khác. Chúng ta sẽ tìm hiểu kỹ hơn về các thành phần này trong phần "Components".

Context-free grammar: Examples

In C, for example, a **while** loop consists of the keyword WHILE followed by a parenthesized Boolean expression and a block of statements:

cÃ¡c

Context-free grammar: Examples

In C, for example, a **while** loop consists of the keyword WHILE followed by a parenthesized Boolean expression and a block of statements:

$$\textit{while\_statement} \longrightarrow \text{WHILE} \; (\; \textit{expression} \;) \; \textit{block\_statement}$$

ký tự ko kết thúc — ký tự kết thúc — kt — ko kt — kt

Context-free grammar: Examples

In C, for example, a **while** loop consists of the keyword WHILE followed by a parenthesized Boolean expression and a block of statements:

Đây là một luật sinh (production rule) được viết theo dạng gần giống Backus-Naur Form (BNF) để mô tả cấu trúc cú pháp của một câu lệnh while (while_statement)

$$\textit{while\_statement} \longrightarrow \text{WHILE} \ (\ \textit{expression} \ ) \ \textit{block\_statement}$$

The statement, in turn, is often a list enclosed in braces:

$$\textit{block\_statement} \longrightarrow \{ \ \textit{statements} \ \}$$

or:

$$\textit{block\_statement} \longrightarrow \{ \ \}$$

Backus-Naur Form (BNF)

1. The notation for context-free grammars is sometimes called **Backus-Naur Form (BNF)**, in honor of John Backus and Peter Naur, who devised it for the definition of the Algol-60 programming language.
2. *Strictly speaking*, RE operators (*, +, ?) and meta-level parentheses are not allowed in BNF, but they do not change the expressive power of the notation, and are commonly included for convenience.

MEng. Tran Ngoc Bao Duy

Backus-Naur Form (BNF)

Context-free grammars (Grammar phi ngữ cảnh): Đây là một loại grammar (ngữ pháp) hình thức mạnh mẽ hơn regular expressions (biểu thức chính quy) và thường được sử dụng để mô tả cú pháp của ngôn ngữ lập trình. Grammar phi ngữ cảnh cho phép chúng ta định nghĩa các cấu trúc lồng nhau (nested structures) trong ngôn ngữ, điều mà regular expressions không làm được một cách hiệu quả.

Backus-Naur Form (BNF): BNF là một kí hiệu (notation) hay cách biểu diễn để viết grammar phi ngữ cảnh. Nói cách khác, BNF là "ngôn ngữ" để mô tả "ngôn ngữ lập trình".

1. The notation for context-free grammars is sometimes called **Backus-Naur Form (BNF)**, in honor of John Backus and Peter Naur, who devised it for the definition of the Algol-60 programming language.
2. *Strictly speaking*, RE operators (*, +, ?) and meta-level parentheses are not allowed in BNF, but they do not change the expressive power of the notation, and are commonly included for convenience.

$\longrightarrow$ **BNF** is used to describe how strings in a language can be derived using **recursive rules**.

Nói một cách chính xác, các toán tử RE (*, +, ?) và dấu ngoặc đơn meta-level không được phép trong BNF, nhưng chúng không làm thay đổi khả năng biểu đạt của kí hiệu, và thường được đưa vào để thuận tiện.

BNF gốc không bao gồm các toán tử RE và dấu ngoặc đơn meta-level. Tuy nhiên, việc thêm chúng vào BNF là phổ biến vì chúng giúp các quy tắc BNF trở nên tiện lợi, dễ đọc và ngắn gọn hơn mà không làm thay đổi khả năng diễn đạt của BNF. Trong thực tế, khi nói đến BNF, người ta thường hiểu là bao gồm cả các tiện ích mở rộng này.

BNF sử dụng các quy tắc đệ quy để mô tả cách tạo ra (dẫn xuất) các chuỗi hợp lệ trong một ngôn ngữ. Kí hiệu $\longrightarrow$ biểu thị một quy tắc sản xuất, và các quy tắc này định nghĩa cấu trúc cú pháp của ngôn ngữ.

*1. Đúng là RE có các kí hiệu +, , ? còn BNF gốc thì không:

Regular Expressions (RE): Đúng vậy, RE chính thức sử dụng các kí hiệu +, *, ? (cùng với nhiều kí hiệu khác như |, [], ()...) để biểu diễn sự lặp lại, tùy chọn, và các mẫu kí tự. Đây là bản chất của sức mạnh biểu đạt ngắn gọn của RE trong việc mô tả các mẫu chuỗi.

Backus-Naur Form (BNF) gốc: BNF nguyên thủy hay thuần túy (pure BNF) được thiết kế ban đầu để định nghĩa cú pháp của ngôn ngữ lập trình Algol-60 mà không có các kí hiệu +, *, ?. Thay vào đó, BNF gốc dựa hoàn toàn vào đệ quy (recursion) để thể hiện sự lặp lại và tùy chọn.

Ví dụ về cách BNF gốc dùng đệ quy thay vì * (lặp lại 0 hoặc nhiều lần):

Giả sử bạn muốn định nghĩa "danh sách các số" (có thể có 0 hoặc nhiều số). Trong RE, bạn có thể dùng number*. Trong BNF gốc, bạn sẽ làm như sau:

BNF

<danh_sach_so> ::= <so> <danh_sach_so_tiep_theo>
<danh_sach_so_tiep_theo> ::= <so> <danh_sach_so_tiep_theo> | /* Rỗng (epsilon) */
<so> ::= "1" | "2" | "3" | ... | "9" | "0"
Giải thích:

<danh_sach_so> được định nghĩa là một <so> theo sau bởi <danh_sach_so_tiep_theo>.
<danh_sach_so_tiep_theo> lại được định nghĩa đệ quy: hoặc là một <so> theo sau bởi <danh_sach_so_tiep_theo> (lặp lại), hoặc là rỗng (kết thúc danh sách hoặc danh sách rỗng ban đầu).
<so> định nghĩa một chữ số.
Như bạn thấy, chúng ta đã dùng đệ quy trong <danh_sach_so_tiep_theo> để tạo ra khả năng lặp lại tương tự như * trong RE, nhưng chỉ dùng quy tắc BNF mà không cần kí hiệu *.

*2. Nếu BNF dùng các kí hiệu +, , ? thì nó khác gì RE?

Đây là điểm quan trọng nhất để làm rõ sự khác biệt về bản chất: *NGAY CẢ KHI BNF CÓ THÊM CÁC KÍ HIỆU +, , ? (BNF mở rộng), NÓ VẪN KHÁC BIỆT HOÀN TOÀN VỚI REGULAR EXPRESSIONS VỀ KHẢ NĂNG BIỂU ĐẠT.

Sự khác biệt nằm ở khả năng xử lý cấu trúc lồng nhau (nested structures) và tính đệ quy mạnh mẽ vốn có của Grammar phi ngữ cảnh (context-free grammar) mà BNF biểu diễn.

Regular Expressions (RE): RE rất mạnh mẽ trong việc mô tả các mẫu tuyến tính (linear patterns) trong chuỗi. Chúng rất tuyệt vời cho việc tìm kiếm, thay thế, và kiểm tra định dạng đơn giản. Tuy nhiên, RE KHÔNG THỂ XỬ LÝ CÁC CẤU TRÚC LỒNG NHAU một cách tự nhiên và hiệu quả.

Ví dụ điển hình: Hãy tưởng tượng bạn muốn kiểm tra một chuỗi có phải là một biểu thức toán học có dấu ngoặc đơn cân bằng hay không (ví dụ: (1+2)*(3+(4)), (), ((())) là hợp lệ, còn (1+2), ((là không hợp lệ). Regular Expressions rất khó khăn (thậm chí không thể một cách thực tế) để làm điều này một cách chính xác và tổng quát. Bạn có thể cố gắng dùng các thủ thuật phức tạp, nhưng sẽ rất rắc rối và không mở rộng được cho các mức độ lồng nhau sâu hơn.
Backus-Naur Form (BNF): BNF (và grammar phi ngữ cảnh nói chung) được thiết kế để mô tả cú pháp của ngôn ngữ lập trình, vốn có bản chất lồng nhau (nested) và đệ quy. BNF có thể dễ dàng mô tả các cấu trúc như:

Biểu thức toán học có dấu ngoặc đơn cân bằng: Bạn có thể dùng quy tắc đệ quy trong BNF để định nghĩa biểu thức có thể chứa biểu thức con bên trong dấu ngoặc đơn, và cứ thế lồng nhau nhiều tầng.
Cấu trúc khối lệnh trong ngôn ngữ lập trình: Ví dụ như vòng if, vòng for, hàm, lớp, chúng đều có cấu trúc lồng nhau (vòng for có thể chứa vòng if, hàm có thể gọi hàm khác...). BNF có thể mô tả các cấu trúc này.
Cấu trúc HTML, XML: Các ngôn ngữ đánh dấu này có thẻ mở và thẻ đóng lồng nhau. BNF rất phù hợp để định nghĩa cú pháp của chúng.
*Tại sao BNF mạnh hơn RE, ngay cả khi có +, , ?:

Ngay cả khi bạn "mở rộng" BNF bằng cách thêm các kí hiệu +, *, ? (như slide đã đề cập là thường được làm để tiện lợi), bản chất đệ quy và khả năng mô tả grammar phi ngữ cảnh của BNF vẫn giữ nguyên và vượt trội hơn hẳn so với RE.

*BNF với +, , ? vẫn là Context-Free Grammar (CFG): Việc thêm các kí hiệu tiện lợi này không biến BNF thành Regular Grammar (loại grammar mà RE mô tả). Nó vẫn thuộc loại Context-Free Grammar, và CFG mạnh hơn Regular Grammar.

Khả năng đệ quy bản chất: Sức mạnh chính của BNF đến từ khả năng đệ quy trong các quy tắc sản xuất. Chính đệ quy cho phép BNF mô tả các cấu trúc lồng nhau mà RE không thể. Các kí hiệu +, *, ? chỉ là "đường tắt" để viết ngắn gọn hơn các quy tắc lặp lại, nhưng chúng không thay đổi khả năng đệ quy cốt lõi của BNF.

Ví dụ so sánh khả năng mô tả dấu ngoặc đơn cân bằng:

BNF (dễ dàng):

BNF

<bieu_thuc> ::= <so> | "(" <bieu_thuc> ")" | <bieu_thuc> "+" <bieu_thuc>
<so> ::= "1" | "2" | ... | "9" | "0"
Quy tắc "(" <bieu_thuc> ")" chính là đệ quy, cho phép biểu thức lồng nhau.

Regular Expressions (vô cùng khó khăn và không thực tế): Bạn sẽ phải dùng các thủ thuật rất phức tạp và vẫn không thể tổng quát cho mọi độ sâu lồng nhau. Về cơ bản, Regular Expressions không đủ sức mạnh để mô tả chính xác ngôn ngữ các chuỗi ngoặc đơn cân bằng.

Tóm lại:

RE (Regular Expressions): Mạnh về mẫu tuyến tính, yếu về cấu trúc lồng nhau, dùng +, *, ? là bản chất. Tốt cho phân tích từ vựng (lexical analysis), tìm kiếm chuỗi đơn giản.
BNF (Backus-Naur Form): Mạnh về cấu trúc lồng nhau, bản chất đệ quy (dù có thể dùng thêm +, *, ? cho tiện lợi). Dùng để định nghĩa cú pháp ngôn ngữ lập trình, mô tả grammar phi ngữ cảnh, cần thiết cho parser (phân tích cú pháp) các ngôn ngữ lập trình phức tạp.

Extended Backus-Naur Form (EBNF)

Yếu tố mới quan trọng nhất mà EBNF mang lại chính là việc giới thiệu các toán tử từ Regular Expressions (RE). Đây là các toán tử mà chúng ta đã thảo luận trước đó: * (Kleene star - lặp lại 0 hoặc nhiều lần), + (Plus - lặp lại 1 hoặc nhiều lần), ? (Question mark - tùy chọn, 0 hoặc 1 lần). Mục đích của việc thêm các toán tử RE là để làm cho việc viết và đọc grammar definitions trở nên đơn giản hơn.

1. **EBNF (Extended Backus-Naur Form)** is an extension of **BNF** that introduces **RE operators** to simplify grammar definitions.
2. EBNF allows for **more compact**, **readable**, and **expressive** grammar rules.
3. It can describe the **same set of context-free languages (CFLs)** as BNF but in a more **succinct way**.

EBNF là một phiên bản nâng cấp của BNF bằng cách thêm các toán tử quen thuộc từ Regular Expressions như *, +, ?. Mục tiêu chính là làm cho việc viết grammar definitions trở nên dễ dàng và trực quan hơn.

Mặc dù về mặt lý thuyết, EBNF và BNF có cùng khả năng biểu đạt (sẽ nói ở điểm 3), nhưng EBNF được coi là biểu đạt hơn theo nghĩa là nó dễ dàng và tự nhiên hơn trong việc diễn đạt ý định của người thiết kế grammar. Các toán tử RE cho phép người viết grammar tập trung vào việc mô tả cấu trúc một cách rõ ràng và trực tiếp, thay vì phải "mã hóa" ý định của mình thành các quy tắc đệ quy phức tạp.

=> EBNF mang lại các ưu điểm về tính ngắn gọn, dễ đọc và khả năng biểu đạt tốt hơn so với BNF gốc, giúp việc thiết kế và hiểu grammar trở nên hiệu quả hơn.

EBNF không làm tăng thêm sức mạnh biểu đạt so với BNF gốc về mặt loại ngôn ngữ mà chúng có thể mô tả. Cả BNF và EBNF đều có khả năng mô tả Context-Free Languages (CFLs) - Ngôn ngữ phi ngữ cảnh.

CFLs là một lớp ngôn ngữ mạnh hơn Regular Languages (ngôn ngữ chính quy - mà RE mô tả). CFLs có thể mô tả các cấu trúc lồng nhau, đệ quy, vốn là đặc trưng của cú pháp ngôn ngữ lập trình. **BNF và EBNF đều là các kí hiệu để định nghĩa CFLs.**

=> EBNF và BNF có cùng sức mạnh biểu đạt, tức là chúng có thể mô tả cùng một tập hợp các ngôn ngữ phi ngữ cảnh (CFLs). EBNF không "mạnh hơn" BNF về mặt loại ngôn ngữ, mà chỉ tiện lợi hơn, ngắn gọn hơn và dễ đọc hơn trong việc mô tả những ngôn ngữ đó. EBNF là một cải tiến về mặt kí hiệu (notational convenience) chứ không phải là một sự gia tăng về sức mạnh lý thuyết.

Extended Backus-Naur Form (EBNF)

EBNF là mở rộng của BNF: Nhằm mục đích làm cho việc viết grammar dễ hơn, ngắn gọn hơn, và dễ đọc hơn.

1. **EBNF (Extended Backus-Naur Form)** is an extension of **BNF** that introduces **RE operators** to simplify grammar definitions.
2. EBNF allows for **more compact**, **readable**, and **expressive** grammar rules.
3. It can describe the **same set of context-free languages (CFLs)** as BNF but in a more **succinct way**.

EBNF dùng các toán tử RE: Chính việc thêm các toán tử *, +, ?, (), [], | (từ RE) là điểm khác biệt lớn nhất và mang lại sự tiện lợi cho EBNF.

Cùng khả năng biểu đạt: BNF và EBNF có thể mô tả cùng một loại ngôn ngữ (CFLs). EBNF chỉ là cách viết "gọn gàng" hơn.

Comparison of BNF and EBNF Features

Feature	BNF	EBNF
Repetition	Recursive rules	*, +, ?
Grouping	Multiple non-terminal rules	Parentheses ()
Alternatives	|	Same, inside ()
Optional Elements	| with ε	Brackets [...]

Lựa chọn (Alternatives): Cả BNF và EBNF đều dùng \| để chọn giữa các khả năng, nhưng EBNF linh hoạt hơn khi cho phép dùng \| bên trong dấu ngoặc đơn để nhóm các lựa chọn lại.

Tùy chọn (Optional Elements):

BNF: Để nói "cái gì đó có thể có hoặc không", phải dùng \| với epsilon (ε) là kí hiệu chuỗi rỗng, ví dụ <phan_nguyen> ::= <chu_so_duong> | ε (phần nguyên có thể có hoặc không). Khó hiểu.
EBNF: Dùng <phan_nguyen> ::= [<chu_so_duong>] (đặt phần tùy chọn trong []). Rất trực quan, giống như "[tùy chọn]".

Tính năng (Feature)	BNF	EBNF	Giải thích ngắn gọn
Lặp lại (Repetition)	**Quy tắc đệ quy (Recursive rules)**	***, +, ?**	- **BNF:** Dùng quy tắc tự tham chiếu (đệ quy) để lặp lại, có thể hơi dài dòng. - **EBNF:** Dùng trực tiếp các toán tử `*` (0+ lần), `+` (1+ lần), `?` (0-1 lần), giống RE, rất ngắn gọn.
Nhóm (Grouping)	**Nhiều quy tắc non-terminal (Multiple non-terminal rules)**	**Dấu ngoặc đơn () (Parentheses)**	- **BNF:** Để nhóm các phần, cần tạo thêm các non-terminal trung gian, làm grammar phức tạp hơn. - **EBNF:** Dùng dấu ngoặc đơn `()` giống như trong toán học hoặc RE để nhóm các thành phần, dễ hiểu.
Lựa chọn (Alternatives)	**|	(or) giống nhau**	**|
Tùy chọn (Optional Elements)	**|	với ε (epsilon - chuỗi rỗng)**	**Dấu ngoặc vuông [ ] (Brackets)**

PARSER: HOW IT WORKS

Nhắc lại:
Lexer có nhiệm vụ:

Đọc mã nguồn gốc dạng văn bản.
Phân tích mã nguồn thành các đơn vị nhỏ hơn có nghĩa (tokens). Ví dụ: nhận diện từ khóa, tên biến, toán tử, số, chuỗi, ...
Gán loại cho mỗi token. Ví dụ: token if có loại KEYWORD, token 123 có loại INTEGER_LITERAL, token + có loại OPERATOR, token count có loại IDENTIFIER, ...
Loại bỏ khoảng trắng, chú thích (comment) không cần thiết.
Đưa ra một chuỗi các token cho Parser xử lý tiếp.

How a Parser Uses CFG

Input (Đầu vào): Parser không làm việc trực tiếp với mã nguồn gốc dạng văn bản. Thay vào đó, đầu vào của Parser là một chuỗi các token.

- **Input:** A sequence of tokens from the lexer.
- **Process:** Coi ở dưới
  1. Start with the **start symbol**.
  2. Apply **production rules** to expand non-terminals.
  3. Match the tokens step by step until the input is fully consumed.
  4. If no valid derivation exists, report a **syntax error**.
- **Output:** A **parse tree** if the input is valid or a **syntax error** if the input violates the grammar.

Output (Đầu ra): Parser có thể tạo ra hai loại đầu ra:
A parse tree if the input is valid (Một cây phân tích cú pháp nếu đầu vào hợp lệ): Nếu Parser phân tích thành công mã nguồn (dẫn xuất thành công chuỗi token từ start symbol và tiêu thụ hết đầu vào), nó sẽ tạo ra parse tree (cây phân tích cú pháp) hoặc Abstract Syntax Tree (AST) - cây cú pháp trừu tượng. Cây này biểu diễn cấu trúc cú pháp của mã nguồn một cách phân cấp và rõ ràng, thể hiện các quan hệ ngữ pháp giữa các thành phần của mã. Parse tree/ AST là đầu vào quan trọng cho các giai đoạn tiếp theo của trình biên dịch/thông dịch (ví dụ: semantic analysis - phân tích ngữ nghĩa, code generation - sinh mã).
a syntax error if the input violates the grammar (một lỗi cú pháp nếu đầu vào vi phạm grammar): Nếu Parser phát hiện lỗi cú pháp (không dẫn xuất được, không tiêu thụ hết đầu vào), nó sẽ báo lỗi cú pháp. Đầu ra trong trường hợp này là thông báo lỗi, không phải parse tree.

Giải thích từng bước trong Process:

Start with the start symbol.Start symbol (Kí hiệu bắt đầu): Trong mỗi grammar (CFG), có một non-terminal đặc biệt được gọi là "start symbol" (kí hiệu bắt đầu). Kí hiệu này thường đại diện cho cấu trúc ngữ pháp lớn nhất, bao trùm toàn bộ chương trình hoặc đơn vị mã nguồn cần phân tích. Ví dụ, start symbol có thể là <program> (chương trình) hoặc <compilation-unit> (đơn vị biên dịch).
"Start with the start symbol": Parser bắt đầu quá trình phân tích từ kí hiệu bắt đầu này. Đây là điểm khởi đầu cho việc xây dựng cây phân tích cú pháp (parse tree).

Apply production rules to expand non-terminals.Apply production rules (Áp dụng các quy tắc sản xuất): Parser sử dụng các quy tắc sản xuất (production rules) đã được định nghĩa trong CFG (ví dụ: bằng BNF/EBNF).
expand non-terminals (mở rộng các non-terminal): Parser sẽ thay thế các non-terminal bằng phần bên phải của các quy tắc sản xuất tương ứng. Quá trình này được gọi là "mở rộng" (expansion) hoặc "dẫn xuất" (derivation). Mục tiêu là dần dần biến đổi kí hiệu bắt đầu thành chuỗi các terminals (token) phù hợp với đầu vào.
Đệ quy (Recursion): Quá trình mở rộng non-terminal có thể đệ quy. Tức là, khi mở rộng một non-terminal, bạn có thể lại gặp các non-terminal khác trong phần bên phải của quy tắc, và lại phải tiếp tục mở rộng chúng. Đây chính là bản chất của việc sử dụng CFG để mô tả cấu trúc lồng nhau.

Match the tokens step by step until the input is fully consumed.Match the tokens step by step (So khớp các token từng bước): Trong quá trình mở rộng non-terminal, Parser sẽ cố gắng so khớp các terminal trong quy tắc sản xuất với các token trong chuỗi đầu vào (từ Lexer). Parser sẽ đọc token đầu vào theo thứ tự.
until the input is fully consumed (cho đến khi đầu vào được tiêu thụ hết): Parser tiếp tục quá trình mở rộng và so khớp cho đến khi tất cả các token trong chuỗi đầu vào đã được "tiêu thụ" (đã được so khớp thành công với các terminal trong quy tắc sản xuất). Nếu Parser "tiêu thụ" hết token đầu vào và thành công trong việc dẫn xuất từ start symbol, điều đó có nghĩa là mã nguồn hợp lệ về mặt cú pháp.

If no valid derivation exists, report a syntax error.If no valid derivation exists (Nếu không có dẫn xuất hợp lệ nào tồn tại): Có thể xảy ra trường hợp Parser không thể tìm được cách áp dụng các quy tắc sản xuất để dẫn xuất ra chuỗi token đầu vào (từ start symbol). Điều này có nghĩa là chuỗi token đầu vào (và do đó, mã nguồn gốc) không tuân thủ đúng ngữ pháp đã định nghĩa.
report a syntax error (báo cáo lỗi cú pháp): Trong trường hợp này, Parser sẽ phát hiện và báo cáo lỗi cú pháp (syntax error). Thông báo lỗi thường bao gồm thông tin về vị trí lỗi (dòng, cột) và mô tả lỗi (ví dụ: "Unexpected token", "Expected ';'", ...). Việc báo lỗi cú pháp là rất quan trọng để giúp lập trình viên sửa lỗi trong mã nguồn.

How a Parser Uses CFG

Ví dụ minh họa quá trình Top-Down Parsing (rất đơn giản hóa):`

Giả sử grammar đơn giản:
<S> ::= <A> <B>
<A> ::= 'a'
<B> ::= 'b'

Chuỗi token đầu vào: a b

Top-down parser hoạt động:

Bắt đầu với start symbol: <S>
Áp dụng quy tắc <S> ::= <A> <B>: Thay <S> bằng <A> <B>
Cần khớp <A> rồi đến <B>.
Mở rộng <A>: Áp dụng quy tắc <A> ::= 'a'
Khớp 'a' với token đầu vào đầu tiên ('a'). Thành công!
Mở rộng <B>: Áp dụng quy tắc <B> ::= 'b'
Khớp 'b' với token đầu vào tiếp theo ('b'). Thành công!
Đã tiêu thụ hết token đầu vào ('a' và 'b'). Dẫn xuất thành công!
Output: Parse tree (cây phân tích cú pháp) thể hiện cấu trúc <S> -> <A> <B> -> 'a' 'b'
Nếu đầu vào là a c

Quá trình sẽ thất bại ở bước 4, vì quy tắc <B> ::= 'b' không khớp với token 'c' trong đầu vào. Parser sẽ báo lỗi cú pháp.

- **Input:** A sequence of tokens from the lexer.
- **Process:**
  1. Start with the **start symbol**.
  2. Apply **production rules** to expand non-terminals.
  3. Match the tokens step by step until the input is fully consumed.
  4. If no valid derivation exists, report a **syntax error**.
- **Output:** A **parse tree** if the input is valid or a **syntax error** if the input violates the grammar.

$\Rightarrow$ This is a **top-down parser** uses context-free grammar to derive the structure of an input string (derivation).

"Top-down" có nghĩa là Parser bắt đầu từ kí hiệu bắt đầu (top - đỉnh) của grammar và cố gắng dẫn xuất (đi xuống) để khớp với chuỗi đầu vào. Có các loại parser khác như "bottom-up parser" (parser từ dưới lên), nhưng slide này tập trung vào top-down.
uses context-free grammar (sử dụng grammar phi ngữ cảnh): Nhấn mạnh lại rằng loại parser này dựa trên Context-Free Grammar (CFG) để hoạt động. CFG cung cấp các quy tắc ngữ pháp mà Parser sử dụng để phân tích.
to derive the structure of an input string (derivation) (để dẫn xuất cấu trúc của một chuỗi đầu vào (dẫn xuất)):
Derivation (dẫn xuất) là quá trình áp dụng các quy tắc sản xuất để biến đổi kí hiệu bắt đầu thành chuỗi token đầu vào. Mục tiêu của parser top-down là thực hiện quá trình dẫn xuất này thành công để xác định cấu trúc cú pháp và tạo ra parse tree.

MEng. Tran Ngoc Bao Duy

Leftmost derivation dẫn xuất trái nhất

Dẫn xuất bên trái là một cách để tạo ra một chuỗi từ một grammar phi ngữ cảnh bằng cách mở rộng non-terminal bên trái nhất ở mỗi bước. Phương pháp này tạo ra một cây phân tích cú pháp bằng cách luôn luôn thay thế non-terminal bên trái nhất trước, đảm bảo một quá trình dẫn xuất có cấu trúc và dễ đoán.

Definition

Leftmost derivation is a way to generate a string from a context-free grammar by expanding the leftmost non-terminal at each step. This method produces a parse tree by always **replacing the leftmost non-terminal first**, ensuring a structured and predictable derivation process.

Điểm mấu chốt của leftmost derivation: Ở MỖI BƯỚC của quá trình dẫn xuất, khi bạn có nhiều non-terminal trong chuỗi hiện tại, bạn LUÔN LUÔN chọn và mở rộng (thay thế) non-terminal NẰM Ở VỊ TRÍ BÊN TRÁI NHẤT.

Ví dụ: Giả sử bạn đang có chuỗi A + B * C, và A, B, C đều là non-terminal. Trong leftmost derivation, bạn sẽ luôn chọn mở rộng A trước, rồi sau đó mới đến B (sau khi có thể đã mở rộng A thành một cái gì đó), và cuối cùng là C. Bạn không được phép chọn mở rộng B hoặc C trước khi đã xử lý xong A (non-terminal bên trái nhất).

Phương pháp leftmost derivation tạo ra một parse tree (cây phân tích cú pháp). Chính việc luôn ưu tiên mở rộng non-terminal bên trái nhất sẽ định hình cấu trúc của parse tree. Mỗi bước mở rộng leftmost derivation tương ứng với một bước xây dựng cây từ trên xuống (top-down) theo hướng từ trái sang phải.

Việc luôn chọn non-terminal bên trái nhất tạo ra một quá trình dẫn xuất có cấu trúc và dễ đoán (predictable). Điều này quan trọng vì nó giúp:

Làm cho quá trình parsing trở nên rõ ràng và hệ thống hơn.
Dễ dàng thực hiện và debug parser.
Đảm bảo tính nhất quán: Với cùng một grammar và cùng một chuỗi đầu vào, leftmost derivation sẽ luôn cho ra cùng một parse tree (nếu chuỗi đó hợp lệ).

MEng. Tran Ngoc Bao Duy

Leftmost derivation

Definition

Leftmost derivation is a way to generate a string from a context-free grammar by expanding the leftmost non-terminal at each step. This method produces a parse tree by always **replacing the leftmost non-terminal first**, ensuring a structured and predictable derivation process.

Benefits of Leftmost Derivation:

tính rõ ràng

1. **Deterministic**: Ensures a clear, step-by-step derivation process.
2. **Easy to Implement**: Forms the basis for top-down parsers.
3. **Predictable Tree Structure:** Guarantees consistent parse tree construction from left to right.

Ví dụ minh họa Leftmost Derivation (ví dụ đơn giản hóa):

Grammar:

S -> AB
A -> a
B -> b
Chuỗi đầu vào: ab

Leftmost Derivation:

Bắt đầu từ S.
Áp dụng quy tắc 1: S => AB (Non-terminal bên trái nhất là A)
Áp dụng quy tắc 2: AB => aB (Non-terminal bên trái nhất còn lại là B)
Áp dụng quy tắc 3: aB => ab (Không còn non-terminal)
Parse Tree (tạo ra từ Leftmost Derivation):

```
  S
 / \
A  B
/  /
a  b
```


MEng. Tran Ngoc Bao Duy

Leftmost derivation: Examples

Grammar:

1. **expr** $\rightarrow$ *expr* $+$ *term* | *term*
2. *term* $\rightarrow$ *term* $*$ *factor* | *factor*
3. *factor* $\rightarrow$ (*expr*) | NUMBER

Leftmost derivation: Examples

Grammar:

expr -> expr + term -> term + term -> factor + term
-> number + term
-> 3 + term
-> 3 + (term * factor)
-> 3 + (factor * factor)
-> 3 + (number * factor)...
... -> 3 + (5 * 2)

1. **expr** $\rightarrow$ *expr* $+$ *term* | *term*
2. *term* $\rightarrow$ *term* $*$ *factor* | *factor*
3. *factor* $\rightarrow$ (*expr*) | NUMBER

Input: $3 + 5 * 2$, derivation (step by Step):

1. ~~Start with the start symbol: **expr**~~ Bắt đầu với kí hiệu bắt đầu expr. expr là start symbol của grammar này, đại diện cho toàn bộ biểu thức.
2. ~~Apply the first rule: **expr** $\rightarrow$ *expr* $+$ *term*~~ Áp dụng quy tắc thứ nhất: expr → expr + term. Đây là quy tắc đầu tiên trong grammar cho expr. Hiện tại ta có: expr + term. Chúng ta đã thay thế expr ban đầu bằng expr + term.
3. ~~Expand the leftmost *expr* again: **expr** $\rightarrow$ *term* + *term*~~
4. ~~Expand the leftmost *term*: *term* $\rightarrow$ *factor*~~
5. ~~Replace *factor* with NUMBER: *factor* $\rightarrow 3$~~

   ~~Now we have: *term* $\rightarrow 3 +$ *term*~~
6. ~~Expand the next *term*: *term* $\rightarrow$ *term* $*$ *factor*~~
7. ~~Replace *term* with *factor*: *term* $\rightarrow$ *factor* $*$ *factor*~~
8. ~~Replace *factor* with NUMBER: *factor* $\rightarrow 5$~~

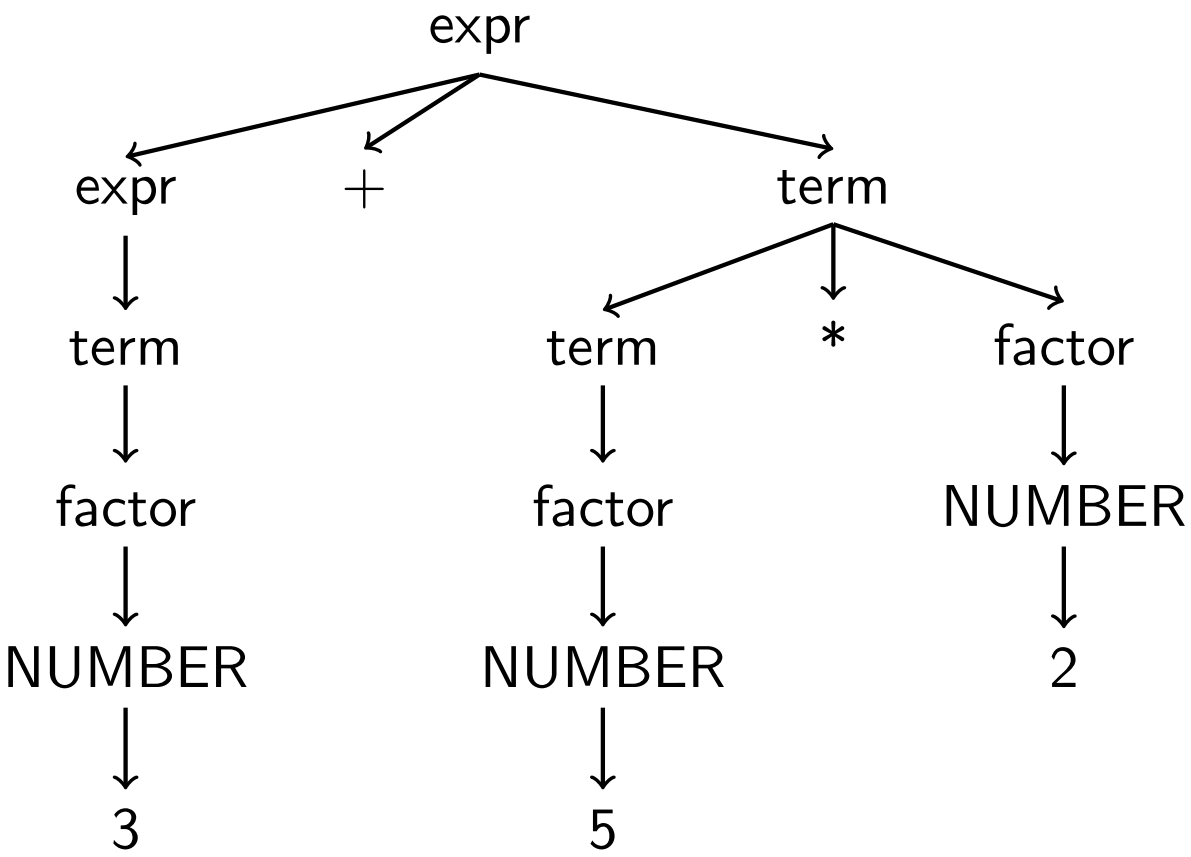
   ~~Now we have: expr $\rightarrow 3 + 5 *$ *factor*~~
9. ~~Replace the last *factor*: *factor* $\rightarrow 2$~~

MEng. Tran Ngoc Bao Duy

Điểm quan trọng cần nhớ:

Trong Leftmost Derivation, chúng ta luôn luôn mở rộng non-terminal bên trái nhất. Tuy nhiên, quy tắc sản xuất cụ thể nào để áp dụng cho non-terminal đó KHÔNG PHẢI là ngẫu nhiên. Quyết định này được dẫn dắt bởi mục tiêu cuối cùng là phải khớp với chuỗi token đầu vào.

Parse Tree as Output

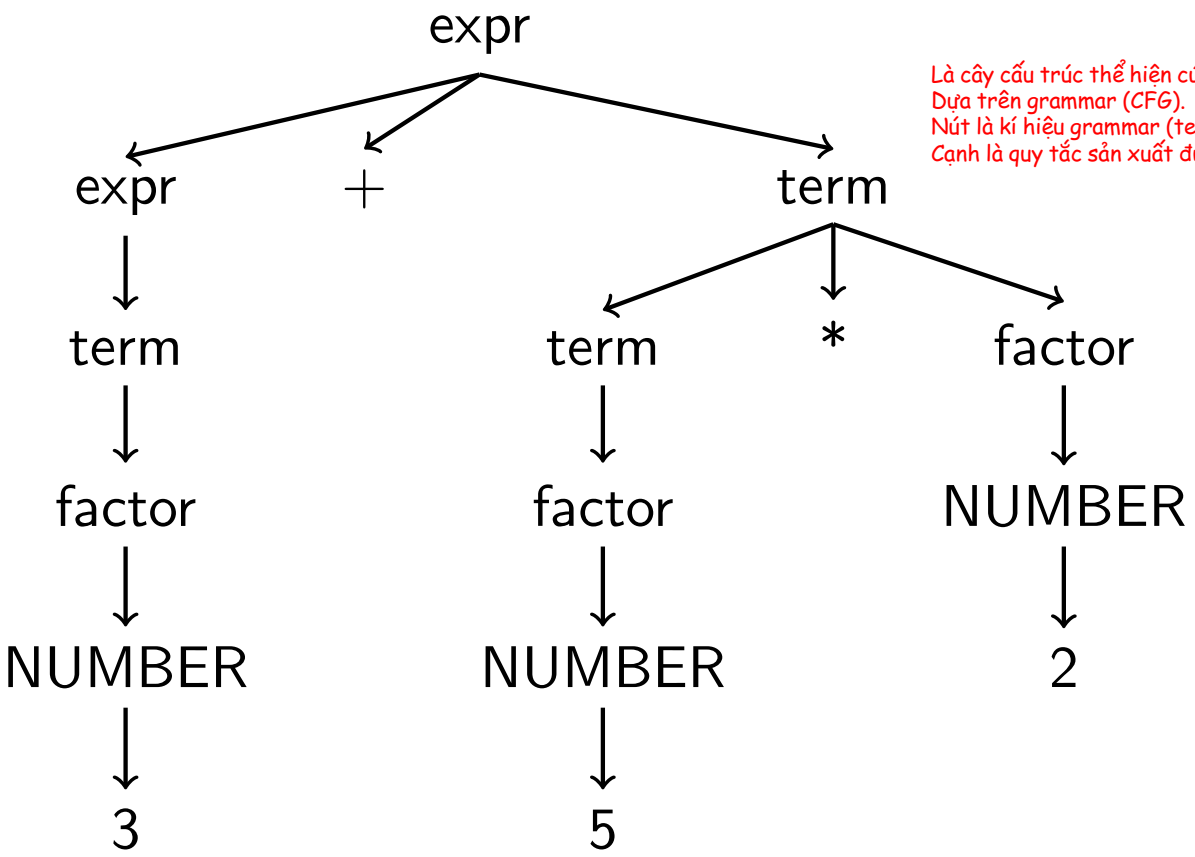


MEng. Tran Ngoc Bao Duy

Parse Tree as Output

Parse Tree không chỉ đơn thuần là danh sách các từ hay token. Nó cho thấy cách các từ và token này kết hợp với nhau theo ngữ pháp để tạo thành một cấu trúc có nghĩa. Nó thể hiện "cấu trúc ngữ pháp" của code, ví dụ như câu lệnh if bao gồm biểu thức điều kiện và khối lệnh bên trong, v.v.

Là cây cấu trúc thể hiện cú pháp của code.
Dựa trên grammar (CFG).
Nút là kí hiệu grammar (terminal hoặc non-terminal).
Cạnh là quy tắc sản xuất được áp dụng.



cấu trúc có phân cấp bieeurr diwwjn có cú phú chuỗi ban đầu của chuỗi ban đầu, nỗi node là 1 ký tự

Definition

A **parse tree** is a hierarchical, tree-like structure that represents the syntactic structure of a string **according to a context-free grammar (CFG)**. Each node in the tree corresponds to a grammar symbol, and the edges represent the application of production rules.

PARSER GENERATION FROM ANTLR4

MEng. Tran Ngoc Bao Duy

ANTLR4: A Quick Recap

Chức năng chính của ANTLR4 là tạo ra parser một cách tự động. Bạn không cần phải viết code parser bằng tay (điều rất phức tạp và dễ mắc lỗi). Thay vào đó, bạn chỉ cần cung cấp cho ANTLR4 một định nghĩa grammar (mô tả cú pháp ngôn ngữ), và ANTLR4 sẽ tự động tạo ra parser.

ANTLR4 không chỉ tạo ra parser, mà còn có thể tạo ra:
Lexer (Bộ phân tích từ vựng): Như chúng ta đã biết, Lexer là thành phần tách mã nguồn thành token. ANTLR4 có thể tạo ra lexer dựa trên grammar bạn cung cấp.
Parser (Bộ phân tích cú pháp): Thành phần chính mà ANTLR4 tạo ra, dùng để phân tích cú pháp dựa trên grammar.
Listener/Visitor code (Mã listener/visitor): Đây là các mẫu thiết kế (design patterns) giúp bạn xử lý và "thăm" (traverse) cây phân tích cú pháp (parse tree) mà parser tạo ra. Listener và Visitor giúp bạn thực hiện các hành động khác nhau trên cây cú pháp, ví dụ: kiểm tra ngữ nghĩa, sinh mã, ...

Recap

ANTLR4 (Another Tool for Language Recognition) is a powerful tool for generating parsers from grammar definitions. It can automatically create **lexer, parser, and listener/visitor code** for various programming languages

Nó có thể tạo ra code parser (và lexer, listener/visitor) bằng nhiều ngôn ngữ lập trình khác nhau, ví dụ: Java, Python, C++, JavaScript, C#, Go, ... Điều này rất tiện lợi vì bạn có thể dùng ANTLR4 để xử lý ngôn ngữ lập trình trong dự án của mình, bất kể bạn đang dùng ngôn ngữ lập trình nào để phát triển dự án.

ANTLR4 Workflow for Parser Generation:

1. **Define the Grammar**: uses a .g4 file, includes lexer rules (for token recognition) and parser rules (for syntactic structure).
2. **Run the command** to generate the parser and lexer code.
3. **Use** the parser and lexer generated.

Định nghĩa grammar trong file .g4 (bao gồm lexer rules và parser rules).
Chạy lệnh ANTLR4 để sinh code parser, lexer.
Sử dụng code đã sinh trong dự án của bạn để phân tích cú pháp ngôn ngữ.

MEng. Tran Ngoc Bao Duy

Writing Parser Rules

Parser rules define how **tokens** are combined to form valid structures.

tên luật sinh or ký hiệu ko kt viết = chữ thường

- Written in **lowercase** (by convention).
- Can reference other parser rules or lexer rules.

về phải

Grammar Features:

Quy tắc parser có thể gọi lẫn nhau. Điều này tạo ra tính đệ quy mạnh mẽ, cho phép định nghĩa các cấu trúc lồng nhau và phức tạp. Ví dụ, quy tắc <expression> có thể tham chiếu đến quy tắc <term>, và <term> có thể tham chiếu đến <factor>.

- Production symbol as :
- Alternatives as |
- Repetition as *, +, ?
- Grouping as ()
- Token end-of-file `EOF` using for starting rule.

Quy tắc parser cũng có thể tham chiếu trực tiếp đến các lexer rules. Điều này có nghĩa là trong quy tắc parser, bạn có thể sử dụng tên token (được định nghĩa trong lexer rules) như thể chúng là một phần của quy tắc parser. Ví dụ, quy tắc parser có thể sử dụng token NUMBER, IDENTIFIER, KEYWORD_IF, ...

Production symbol as :: Trong ANTLR4 grammar file (.g4), dấu hai chấm : được sử dụng làm kí hiệu sản xuất trong quy tắc parser. Nó tương tự như → hoặc ::= trong BNF/EBNF, nhưng ANTLR4 dùng : để đơn giản. Ví dụ: ruleName : ruleBody;

Token end-of-file EOF using for starting rule: EOF (End-Of-File - Kết thúc file) là một token đặc biệt, tự động được thêm vào cuối chuỗi token đầu vào sau khi lexer xử lý xong toàn bộ mã nguồn.
using for starting rule: Trong ANTLR4, quy ước tốt là quy tắc parser bắt đầu (start rule) (thường đại diện cho toàn bộ chương trình hoặc đơn vị biên dịch) nên "tiêu thụ" token EOF ở cuối. Điều này đảm bảo rằng parser đã xử lý hết toàn bộ input và không còn token "thừa" nào ở cuối file. Ví dụ: program : statement+ EOF; (Chương trình là một hoặc nhiều câu lệnh, theo sau bởi EOF).

MEng. Tran Ngoc Bao Duy

Example Grammar: Arithmetic Language

```
grammar Arithmetic;

// Parser Rules
expr: expr '+' term   // Addition
    | expr '-' term   // Subtraction
    | term;           // Single Term

term: term '*' factor   // Multiplication
    | term '/' factor   // Division
    | factor;           // Single Factor

factor: '(' expr ')'    // Parentheses
      | NUMBER;         // Numbers

// Lexer Rules
NUMBER: [0-9]+;             // Integer
WS: [ \t\r\n]+ -> skip;   // Ignore whitespace
```


CHALLENGES IN GRAMMAR WRITING

MEng. Tran Ngoc Bao Duy

Ambiguous Grammar in CFGs Ngữ pháp mơ hồ trong CFGs

gây ra đa nghĩa trong cú pháp. "CFGs" nhắc lại rằng chúng ta đang nói về Context-Free Grammars.

Definition văn phạm phi ngữ cảnh tối nghĩa có ít nhất 2 cây parse tree tạo ra cho 1 chuỗi

A context-free grammar (CFG) is said to be **ambiguous** if there exists at least one string that can have **more than one distinct parse tree** or derivation dẫn xuất. This means the grammar allows multiple valid interpretations for the same input, leading to syntactic ambiguity.

- **Multiple Parse Trees:** A single input string can lead to more than one valid parse tree.
- **Parser Conflicts:** Ambiguous grammars often cause conflicts during parsing.
- **Complex Disambiguation:** Resolving ambiguity typically requires modifying the grammar to enforce precedence, associativity, or restructuring rules.

Một grammar phi ngữ cảnh (CFG) được gọi là mơ hồ nếu tồn tại ít nhất một chuỗi mà có thể có nhiều hơn một cây phân tích cú pháp hoặc dẫn xuất khác biệt. Điều này có nghĩa là grammar cho phép nhiều cách hiểu hợp lệ cho cùng một đầu vào, dẫn đến tính mơ hồ cú pháp:

Distinct parse tree (Cây phân tích cú pháp khác biệt): Có ít nhất hai parse tree khác nhau về cấu trúc (dù có thể cho ra cùng một kết quả về mặt ngữ nghĩa, nhưng cấu trúc phân tích khác nhau).

or derivation (hoặc dẫn xuất khác biệt): Hoặc có ít nhất hai leftmost (hoặc rightmost) derivation khác nhau để tạo ra cùng một chuỗi. (Thường thì có nhiều parse tree sẽ tương ứng với nhiều derivation).

=> Điều này gây ra vấn đề vì parser không biết nên chọn cách hiểu nào là đúng.

Multiple Parse Trees (Nhiều Cây Phân tích Cú pháp): Đây là biểu hiện trực quan của sự mơ hồ. Nếu bạn dùng một grammar mơ hồ để phân tích một chuỗi, bạn có thể vẽ ra ít nhất hai parse tree khác nhau, và cả hai cây này đều hợp lệ theo grammar. Sự tồn tại của nhiều parse tree là dấu hiệu rõ ràng của ngữ pháp mơ hồ.

Parser Conflicts (Xung đột Parser): Khi bạn sử dụng một parser generator (như ANTLR4) để tạo parser từ một ngữ pháp mơ hồ, hoặc khi bạn cố gắng viết một parser cho ngữ pháp mơ hồ, bạn sẽ thường gặp phải "parser conflicts" (xung đột parser). Xung đột này xảy ra vì parser không biết nên chọn quy tắc sản xuất nào khi có nhiều hơn một lựa chọn hợp lệ tại một bước nào đó trong quá trình phân tích. Các loại xung đột phổ biến như "shift-reduce conflict" và "reduce-reduce conflict".

Khử Mơ hồ Phức tạp: Việc giải quyết sự mơ hồ thường đòi hỏi phải sửa đổi grammar để ép buộc độ ưu tiên, tính kết hợp, hoặc tái cấu trúc các quy tắc.

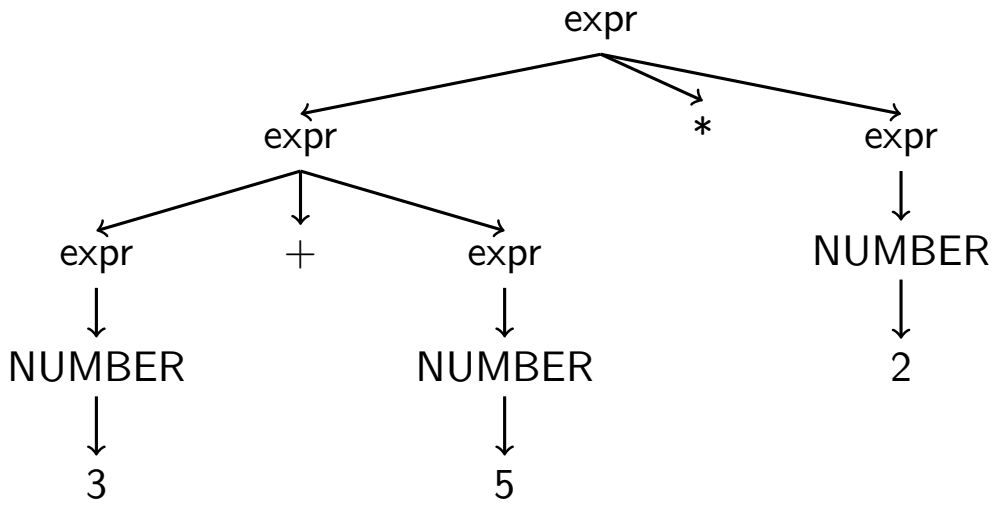
Ambiguous Grammar: Example

Grammar (in ANTLR4):

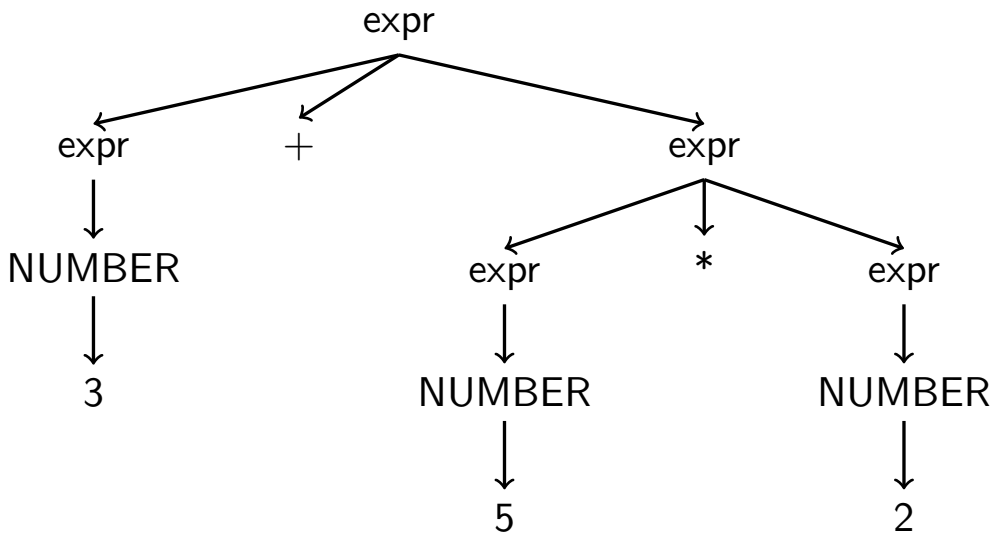
```
grammar AmbiguousExpr;

expr: expr ('+' | '-') expr    // AddSub
    | expr ('*' | '/') expr    // MulDiv
    | '(' expr ')'             // Parens
    | NUMBER;                  // Number

NUMBER: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

For an input $3 + 5 * 2$, there are two possible parse trees

Example: two possible parse trees



Parse Tree 1: $(3 + 5) * 2$



Parse Tree 2: $3 + (5 * 2)$

MEng. Tran Ngoc Bao Duy

Common Patterns in Writing Parser Rules

Trong quá trình viết quy tắc parser, cách tiếp cận thông thường là tuân theo thứ tự mà chúng xuất hiện trong đặc tả ngôn ngữ. Tuy nhiên, trong quá trình viết, có năm mẫu lặp lại thường xuyên xuất hiện

In the process of writing parser rules, the typical approach is to follow the order in which they appear in the language specification. However, during the writing process, there are **five recurring patterns** that frequently appear:

1. A non-empty list of x.
2. A null-able list of x.
3. A non-empty list of x, separated by y.
4. A null-able list of x, separated by y.
5. An infix expression.

Mẫu này mô tả một cấu trúc ngữ pháp mà yêu cầu phải có ít nhất một hoặc nhiều phần tử (x) trong danh sách.

Mẫu này mô tả cấu trúc ngữ pháp mà danh sách có thể trống rỗng, tức là có thể có 0 hoặc nhiều phần tử (x).

Mẫu này kết hợp mẫu 1 (danh sách không rỗng) và thêm dấu phân tách (y) giữa các phần tử. Yêu cầu có ít nhất một 'x', và nếu có nhiều 'x', chúng phải được phân tách bởi 'y'.

giống ở trên, nhưng nullable

Mẫu này mô tả biểu thức toán học hoặc logic mà toán tử nằm giữa các toán hạng (operand), ví dụ: a + b, 5 * 2, x > y, a && b.
Infix expression: Là dạng biểu thức phổ biến trong toán học và nhiều ngôn ngữ lập trình. Viết parser cho infix expression thường phức tạp hơn một chút do cần xử lý độ ưu tiên của toán tử (operator precedence) và tính kết hợp (associativity) (như đã thảo luận ở các slide trước về ngữ pháp mơ hồ).
Ví dụ: Biểu thức số học (ví dụ: 3 + 5 * 2), biểu thức logic (ví dụ: a && b || !c).

Case 1: A non-empty list of x

Example

A **program** in the C language consists of at least one **function**. The non-terminal symbol for a program is *program* (as a start symbol), and for a *func*, it is function.
Write the production rule for program.

main()

Explanation: A **program** in the C language consists of at least one **function** $\Rightarrow$ A **program** in the C language consists of a non-empty list of **functions**.

Slide diễn giải lại ràng buộc "ít nhất một hàm" thành khái niệm "danh sách không rỗng các hàm". Điều này rất quan trọng để chúng ta hình dung ra cách viết quy tắc grammar.

BNF: program ::= function | program function
EBNF: program : function+ ;

Case 1: A non-empty list of x

Example

A **program** in the C language consists of at least one **function**. The non-terminal symbol for a program is *program* (as a start symbol), and for a *func*, it is function.
Write the production rule for program.

1 danh sách các function ko đc rỗng

Explanation: A **program** in the C language consists of at least one **function** ⇒ A **program** in the C language consists of a non-empty list of **functions**.

- With BNF:

```
program: func_list EOF;
func_list: func func_list | func;
```

- With EBNF:

```
program: func_list EOF;
func_list: func+;
```


Formula 1: A non-empty list of x

Every case in the format of **a non-empty list of x** can be written using the following formula:

1. With BNF:

```
xlist: x xlist | x;
```

2. With EBNF:

```
xlist: x+;
```


Case 2: A null-able list of x

Example

The **body of a function** in C consists of a null-able list of **statements** enclosed in a pair of curly braces. The function body is denoted as *func_body*, and a statement is denoted as *stmt*.
Write the production rule for the function body.

MEng. Tran Ngoc Bao Duy

Case 2: A null-able list of x

Example

The **body of a function** in C consists of a null-able list of **statements** enclosed in a pair of curly braces. The function body is denoted as *func_body*, and a statement is denoted as *stmt*.
Write the production rule for the function body.

- With BNF:

```
func_body: '{' stmtlist '}';
stmtlist: stmt stmtlist | ;
```

- With EBNF:

```
func_body: '{' stmtlist '}';
stmtlist: stmt*;
```


Formula 2: A non-empty list of x

Every case in the format of **a null-able list of x** can be written using the following formula:

1. With BNF:

```
xlist: x xlist | ;
```

2. With EBNF:

```
xlist: x*;
```


Case 3: A non-empty list of x, separated by y

Example

A **variable declaration** starts with a **type**, which is int or float, then **a comma-separated list of identifiers** and ends with a semicolon. It is known that a variable declaration is denoted as *vardecl*, and an identifier is denoted as ID.
Other terminal symbols include: INT (integer type), FLOAT (floating-point type), SEMI (semicolon), and CM (comma).
Write the production rule for the variable declaration.

Case 3: A non-empty list of x, separated by y

Example

A **variable declaration** starts with a **type**, which is int or float, then **a comma-separated list of identifiers** and ends with a semicolon. It is known that a variable declaration is denoted as *vardecl*, and an identifier is denoted as ID.
Other terminal symbols include: INT (integer type), FLOAT (floating-point type), SEMI (semicolon), and CM (comma).
Write the production rule for the variable declaration.

- With BNF:

```
vardecl: (INT | FLOAT) idlist SEMI;
idlist: ID CM idlist | ID;
```

- With EBNF:

```
vardecl: (INT | FLOAT) idlist SEMI;
idlist: ID (CM ID)*;
```


Formula 3: A non-empty list of x, separated by y

Every case in the format of **a non-empty list of x, separated by y** can be written using the following formula:

1. With BNF:

```
xlist: x y xlist | x;
```

2. With EBNF:

```
xlist: x (y x)*;
```


Case 4: A null-able list of x, separated by y

Example

The **parameter declaration** starts with a left round bracket LB and **a null-able semicolon-separated list of parameters** and ends with a right round bracket RB. It is known that a parameter declaration is denoted as *paramdecl*, and a parameter is denoted as *param*.
Write the production rule for the parameter declaration.

Case 4: A null-able list of x, separated by y

Example

The **parameter declaration** starts with a left round bracket LB and **a null-able semicolon-separated list of parameters** and ends with a right round bracket RB. It is known that a parameter declaration is denoted as *paramdecl*, and a parameter is denoted as *param*.
Write the production rule for the parameter declaration.

- With BNF:

```
paramdecl: LB paramlist RB;
paramlist: param SM paramlist | ;
```

- With EBNF:

```
paramdecl: LB paramlist RB;
paramlist: param (SM param)* | ;
```


Case 4: A null-able list of x, separated by y

Example

The **parameter declaration** starts with a left round bracket LB and **a null-able semicolon-separated list of parameters** and ends with a right round bracket RB. It is known that a parameter declaration is denoted as *paramdecl*, and a parameter is denoted as *param*.
Write the production rule for the parameter declaration.

- With BNF:

```
paramdecl: LB paramlist RB;
paramlist: param SM paramlist | ;
```

- With EBNF:

```
paramdecl: LB paramlist RB;
paramlist: param (SM param)* | ;
```

Is the solution above correct?

Case 4: A null-able list of x, separated by y

Example

The **parameter declaration** starts with a left round bracket LB and **a null-able semicolon-separated list of parameters** and ends with a right round bracket RB. It is known that a parameter declaration is denoted as *paramdecl*, and a parameter is denoted as *param*.
Write the production rule for the parameter declaration.

- With BNF:

```
paramdecl: LB paramlist RB;
paramlist: param SM paramlist | ;
```

- With EBNF:

```
paramdecl: LB paramlist RB;
paramlist: param (SM param)* | ;
```

Is the solution above correct? **NO**

Đối với một số thuật toán parser, đặc biệt là Recursive Descent Parser (parser đệ quy xuống), left recursion có thể gây ra vòng lặp vô hạn. Khi parser cố gắng phân tích paramlist, nó sẽ ngay lập tức gọi lại chính paramlist mà chưa "tiêu thụ" bất kỳ token đầu vào nào, dẫn đến lặp vô hạn.
Mặc dù về mặt lý thuyết, grammar left-recursive vẫn có thể định nghĩa ngôn ngữ, nhưng chúng không phù hợp với các parser top-down đơn giản.

Case 4: A null-able list of x, separated by y

Example

The **parameter declaration** starts with a left round bracket LB and **a null-able semicolon-separated list of parameters** and ends with a right round bracket RB. It is known that a parameter declaration is denoted as *paramdecl*, and a parameter is denoted as *param*.
Write the production rule for the parameter declaration.

- With BNF:

```
paramdecl: LB paramlist RB;
paramlist: paramprime | ;
paramprime: param SM paramprime | param;
```
ít nhất 1 phần tử

- With EBNF:

```
paramdecl: LB paramlist RB;
paramlist: param (SM param)* | ;
```

or `paramlist: (param (SM param)*)?;`

Đoạn BNF đầu tiên cho paramlist bị coi là "KHÔNG ĐÚNG" (NO) vì nó left-recursive, có thể gây vấn đề cho một số parser top-down đơn giản.
Đoạn BNF thứ hai (đã sửa) được coi là "ĐÚNG" (YES) vì nó sử dụng right recursion thông qua paramprime, tránh được vấn đề của left recursion và vẫn định nghĩa cùng một ngôn ngữ.
Các phiên bản EBNF được coi là "ĐÚNG" (YES) vì EBNF sử dụng toán tử lặp * (và ?), giúp biểu diễn danh sách một cách ngắn gọn và hiệu quả, ẩn đi cơ chế đệ quy và không gây ra vấn đề vòng lặp. Về mặt ngôn ngữ được định nghĩa, các phiên bản EBNF (cả "ban đầu" và "gọn hơn") đều tương đương và "đúng".

BNF "KHÔNG ĐÚNG" (ban đầu):
paramlist : param SM paramlist | ;

BNF "ĐÚNG" (đã sửa):
paramlist : paramprime | ;
paramprime : param SM paramprime | param ;

Sự khác biệt chính và ngắn gọn:
BNF "KHÔNG ĐÚNG" (ban đầu): Sử dụng đệ quy trái (left recursion) trực tiếp trong quy tắc paramlist. Điều này có nghĩa là khi muốn phân tích paramlist, parser sẽ cố gắng nhận diện param, sau đó lại gọi chính paramlist ngay lập tức ở đầu quy tắc. Kiểu đệ quy này có thể gây ra vòng lặp vô hạn cho các parser top-down (như Recursive Descent Parser).

BNF "ĐÚNG" (đã sửa): Loại bỏ đệ quy trái bằng cách giới thiệu một quy tắc trung gian mới là paramprime. Quy tắc paramprime sử dụng đệ quy phải (right recursion). Parser bây giờ sẽ nhận diện param, sau đó mới gọi lại paramprime ở cuối quy tắc. Đệ quy phải không gây ra vấn đề vòng lặp cho parser top-down. Quy tắc paramlist sau đó chỉ đơn giản là chọn giữa paramprime (danh sách không rỗng) hoặc chuỗi rỗng (danh sách rỗng).

Điểm mấu chốt:
Cả hai định nghĩa BNF về mặt lý thuyết có thể mô tả cùng một ngôn ngữ (tức là cùng một tập hợp các chuỗi hợp lệ cho paramlist). Sự khác biệt nằm ở cách thực hiện phân tích cú pháp.
BNF "KHÔNG ĐÚNG" (đệ quy trái): Không phù hợp cho các parser top-down vì có thể dẫn đến vòng lặp vô hạn trong quá trình phân tích.
BNF "ĐÚNG" (đệ quy phải): Phù hợp với parser top-down vì tránh được vấn đề vòng lặp và vẫn đảm bảo mô tả đúng cấu trúc danh sách tham số.

Formula 4: A null-able list of x, separated by y

Every case in the format of **a null-able list of x, separated by y** can be written using the following formula:

1. With BNF:

```
xlist: xprime | ;
xprime: x y xprime | x;
```

2. With EBNF:

```
xlist: x (y x)* | ;
```

or:

```
xlist: (x (y x)*)?;
```


Case 5: An infix expression

"Please Excuse My Dear Aunt Sally" (PEMDAS) là một quy tắc nhớ phổ biến được dùng để ghi nhớ thứ tự các phép toán trong toán học. Mỗi từ trong cụm từ đại diện cho một phép toán cụ thể, được liệt kê theo thứ tự chúng nên được thực hiện:

P - Dấu ngoặc đơn (Giải các biểu thức bên trong dấu ngoặc đơn trước)
E - Lũy thừa (Tính lũy thừa và căn thức tiếp theo)
MD - Nhân và Chia (Từ trái sang phải)
AS - Cộng và Trừ (Từ trái sang phải)"

"Please Excuse My Dear Aunt Sally"(PEMDAS) is a popular mnemonic used to remember the **order of operations** in mathematics. Each word in the phrase represents a specific mathematical operation, listed in the order they should be performed:

- P – Parentheses (Solve expressions inside parentheses first)
- E – Exponents (Calculate powers and roots next)
- MD – Multiplication and Division (From left to right)
- AS – Addition and Subtraction (From left to right)

In programming (as mathematics), **infix expressions** require explicit rules for **the order of operations** (such as PEMDAS), while **prefix** and **postfix** expressions do not.

Trong lập trình (cũng như toán học), biểu thức infix đòi hỏi các quy tắc rõ ràng cho thứ tự phép toán (như PEMDAS), trong khi biểu thức prefix và postfix thì không.

Ví dụ về sự khác biệt:

Infix: 3 + 5 * 2 (cần PEMDAS để hiểu là 3 + (5 * 2) chứ không phải (3 + 5) * 2)
Prefix: + 3 * 5 2 (Không mơ hồ, rõ ràng là cộng 3 với kết quả của nhân 5 và 2)
Postfix: 3 5 2 * + (Không mơ hồ, rõ ràng là 5 * 2 rồi cộng với 3)

MEng. Tran Ngoc Bao Duy

Case 5: An infix expression - Example

Given a piece of grammar defined in ANTLR as below:

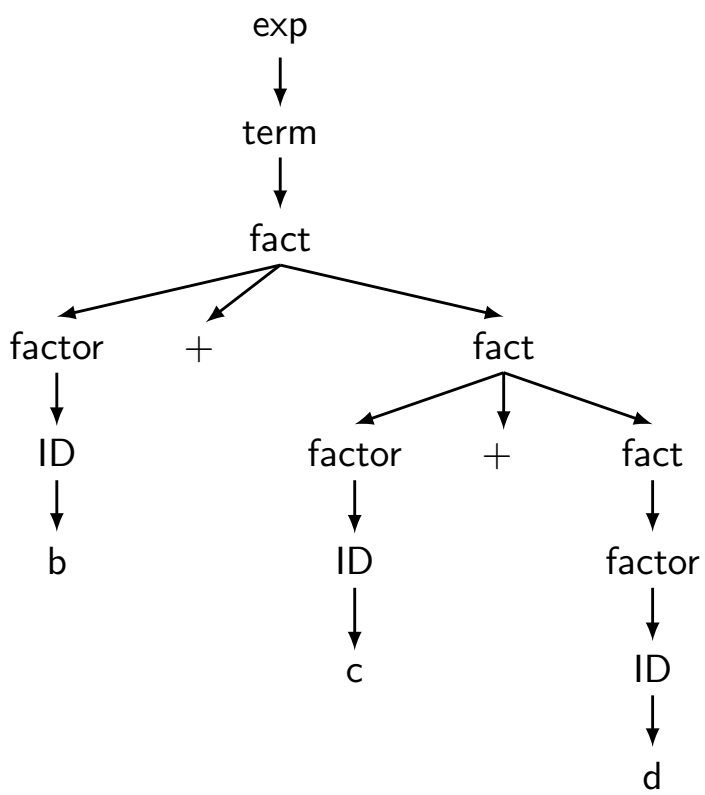
```
grammar exp;
exp      : term ASSIGN exp | term;
term     : term EXPONENT fact | fact;
fact     : factor RELOP fact | factor ADDOP fact
         | factor;
factor   : ID;
ASSIGN   : '=' ;
EXPONENT : '^' ;
ADDOP    : '+' ;
RELOP    : '>' ;
ID       : [a-zA-Z_][a-zA-Z_0-9]* ;
WS       : [ \t\r\n]+ -> skip ;
```

Quy tắc nào được định nghĩa "sâu hơn" (gần factor hơn) sẽ có độ ưu tiên cao hơn. Ví dụ: phép lũy thừa (^) được xử lý trong term, quy tắc term được gọi từ exp, và term lại gọi fact. Do đó, ^ có độ ưu tiên cao hơn phép gán = (xử lý trong exp). Tương tự (nhưng không chính xác trong ví dụ này), slide có vẻ muốn ám chỉ RELOP và ADDOP trong fact có độ ưu tiên cao hơn = và ^ (điều này không đúng với thứ tự PEMDAS chuẩn và grammar ví dụ cũng không thực sự thể hiện điều này một cách chính xác).

MEng. Tran Ngoc Bao Duy

Case 5: An infix expression - Example

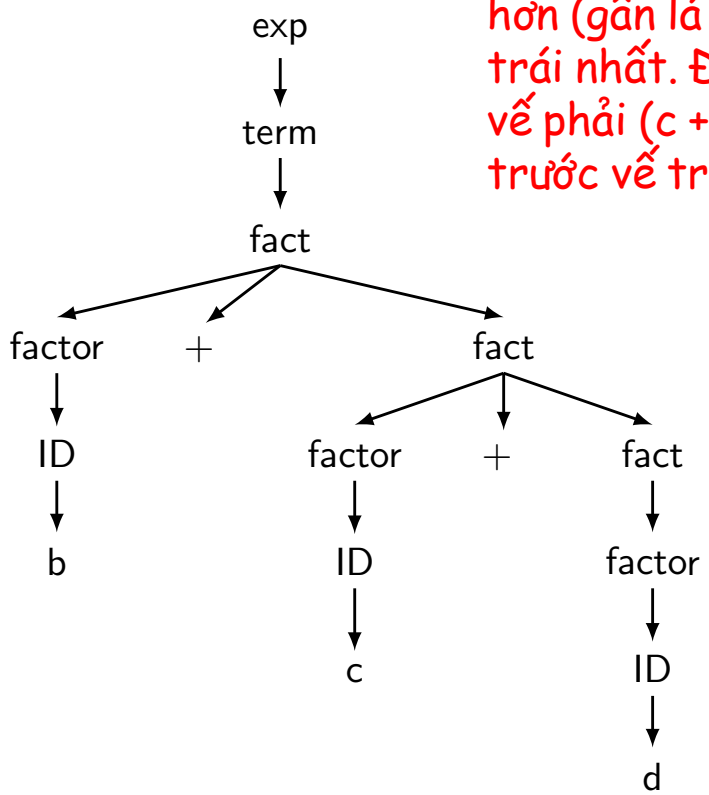
Parse Tree for `b + c + d`:



MEng. Tran Ngoc Bao Duy

Case 5: An infix expression - Example

Parse Tree for `b + c + d`:



Trong cây này, dấu + bên phải nhất (giữa c và d) xuất hiện sâu hơn (gần lá hơn) so với dấu + bên trái nhất. Điều này chỉ ra rằng vế phải (c + d) được tính toán trước vế trái (b + kết quả)

In this tree, the rightmost + (between c and d) appears deeper (closer to the leaves) than the leftmost one. This indicates that the **right** side (c + d) is **evaluated before** the **left** (b + result).

Cách diễn giải Right-associativity qua Parse Tree (theo hướng slide muốn):

"Rightmost + (between c and d) appears deeper..." - Như slide đã chỉ ra, dấu + giữa c và d ở nhánh sâu hơn. Nếu chúng ta diễn giải theo hướng right-associativity, điều này có nghĩa là phép cộng c + d được thực hiện trước, và kết quả sau đó được cộng vào b. Tức là b + (c + d).
"...This indicates that the right side (c + d) is evaluated before the left (b + result)." - Câu này, trong ngữ cảnh của right-associativity, mới có ý nghĩa đúng đắn. Nó khẳng định rằng (c + d) (vế phải) được tính trước, và sau đó b (vế trái) được cộng vào kết quả của (c + d).
Tuy nhiên, vẫn cần lưu ý những điểm quan trọng và có thể gây nhầm lẫn:

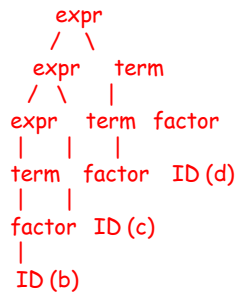
Phép cộng trong toán học và hầu hết các ngôn ngữ lập trình THƯỜNG LÀ LEFT-ASSOCIATIVE (kết hợp từ trái sang phải), KHÔNG PHẢI RIGHT-ASSOCIATIVE. Quy tắc PEMDAS/BODMAS cũng chỉ rõ "Addition and Subtraction (From left to right)". Việc slide khẳng định ADDOP (phép cộng) là right-associative có thể là một ví dụ tương phản hoặc một trường hợp đặc biệt để minh họa khái niệm, chứ không phải là quy tắc chung cho phép cộng.

Cấu trúc Parse Tree trong hình ảnh vẫn có thể hơi "lạ" nếu diễn giải theo right-associativity. Thông thường, cây parse tree cho phép toán kết hợp phải (right-associative) sẽ có cấu trúc "xuống dần về phía bên phải", chứ không phải cấu trúc có vẻ "cân đối" như trong hình ảnh này.

Để thực sự khẳng định ADDOP là right-associative và Parse Tree này thể hiện điều đó, chúng ta cần xem xét grammar cụ thể đã sinh ra cây này. Chính grammar định nghĩa tính kết hợp của toán tử, và Parse Tree chỉ là kết quả của việc phân tích chuỗi theo grammar đó. Nếu grammar định nghĩa ADDOP là right-associative, thì Parse Tree này có thể là một cách biểu diễn hợp lệ (dù có thể không phải là cách biểu diễn phổ biến nhất cho right-associativity).

Case 5: An infix expression - Example

(Cây này có thể khác với cây trong hình ảnh một chút về cấu trúc nhánh, nhưng thể hiện tính kết hợp từ trái sang phải rõ hơn)



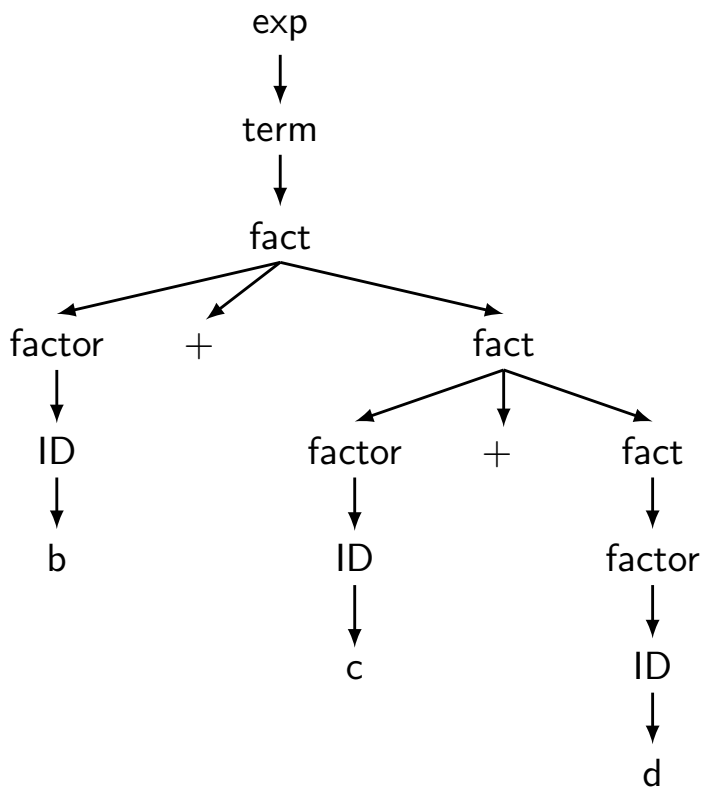
Parse Tree for b + c + d:

In this tree, the rightmost + (between c and d) appears deeper (closer to the leaves) than the leftmost one. This indicates that the **right** side (c + d) is **evaluated before** the **left** (b + result).

→ The ADDOP is right-associative.

Case 5: An infix expression - Example

Parse Tree for `b + c + d`:




```
fact: factor ADDOP fact | factor ;
```

- This is **right-recursive** because *fact* appears on the **right-hand side** of the production rule.

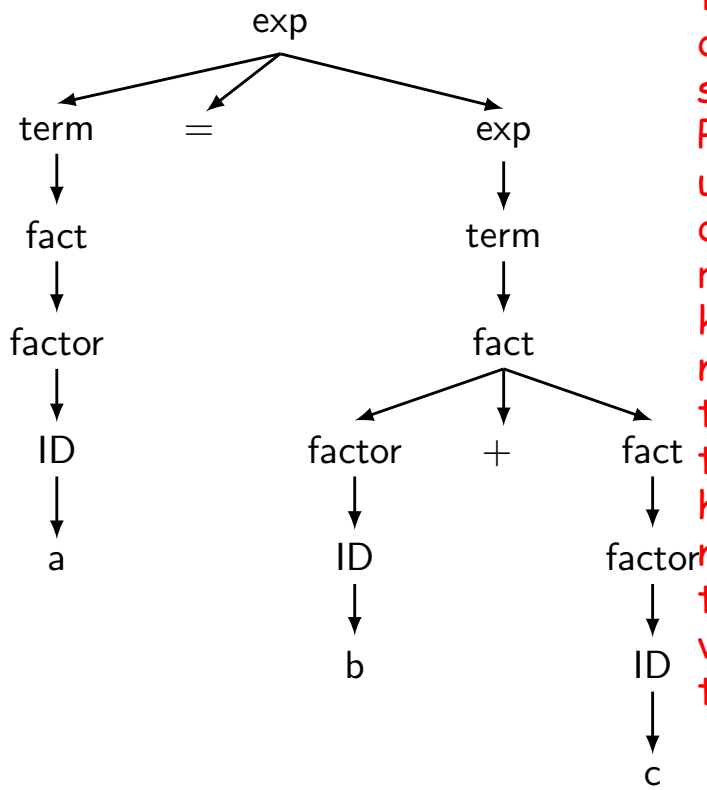
kết hợp phải

- **Right recursion** leads to **right-associative** behavior.

đệ quy nằm bên phải dấu

Case 5: An infix expression - Example

Parse Tree for `a = b + c`



Các kết luận này, khi được xem xét cùng nhau, mô tả cách grammar right-recursive có thể dẫn đến tính kết hợp từ phải sang trái cho toán tử (trong ví dụ là ADDOP), và cách độ sâu của toán tử trong Parse Tree phản ánh độ ưu tiên của chúng. Mặc dù ví dụ về phép cộng right-associative có thể không điển hình, các nguyên tắc về đệ quy, tính kết hợp và độ ưu tiên, và cách chúng thể hiện trên Parse Tree, là những khái niệm rất quan trọng trong việc thiết kế và hiểu grammar cho biểu thức infix.

- ASSIGN (=) is **higher** in the tree, giving it lower precedence.
- The **depth of operators** in the parse tree directly reflects **their precedence**.

MEng. Tran Ngoc Bao Duy

Formula 5: An infix expression

1. **Operator Precedence:** The closer an operator's production rule is to the start production rule, the lower its precedence.
2. **Operator Associativity**:
   - **Left-recursive** rules enforce **left-associativity**.
   - **Right-recursive** rules enforce **right-associativity**.
   - **Non-recursive** rules enforce **non-associativity**.

"The closer an operator's production rule is to the start production rule...": Nguyên tắc này nói về vị trí tương đối của quy tắc sản xuất cho một toán tử trong toàn bộ grammar. Grammar thường có một start production rule (quy tắc sản xuất bắt đầu), thường là quy tắc cao nhất, bao trùm toàn bộ cấu trúc ngôn ngữ (ví dụ: program, expression, statement). Các quy tắc khác được định nghĩa để xây dựng nên các thành phần con của ngôn ngữ.

"...the lower its precedence.": Mối quan hệ nghịch đảo giữa vị trí quy tắc và độ ưu tiên:

Quy tắc sản xuất cho toán tử được định nghĩa gần với start production rule (ở mức cao hơn trong cấu trúc grammar): Toán tử đó có độ ưu tiên thấp hơn. Nó sẽ được thực hiện sau các toán tử có độ ưu tiên cao hơn.
Quy tắc sản xuất cho toán tử được định nghĩa xa start production rule hơn (ở mức thấp hơn trong cấu trúc grammar, thường được gọi từ các quy tắc "cao hơn"): Toán tử đó có độ ưu tiên cao hơn. Nó sẽ được thực hiện trước các toán tử có độ ưu tiên thấp hơn.

expr ::= term ('+' term)* // Quy tắc expr (cao nhất, gần start rule) - chứa phép cộng (+)
term ::= factor ('*' factor)* // Quy tắc term (ở giữa) - chứa phép nhân (*)
factor ::= '(' expr ')' | NUMBER // Quy tắc factor (thấp nhất, xa start rule hơn) - chứa dấu ngoặc đơn

expr (gần start rule hơn, cao nhất): Chứa phép cộng + (độ ưu tiên thấp nhất trong ví dụ này).
term (ở giữa): Chứa phép nhân * (độ ưu tiên trung bình, cao hơn + nhưng thấp hơn dấu ngoặc).
factor (xa start rule hơn, thấp nhất trong cấu trúc quy tắc): Chứa dấu ngoặc đơn () (độ ưu tiên cao nhất).

Kết luận: Càng "xuống sâu" trong cấu trúc grammar (từ expr -> term -> factor), độ ưu tiên của toán tử càng tăng lên. Quy tắc của toán tử ở càng "xa" start rule hơn, độ ưu tiên càng cao.

• Quy tắc đệ quy trái ép buộc tính kết hợp từ trái sang phải.
• Quy tắc đệ quy phải ép buộc tính kết hợp từ phải sang trái.
• Quy tắc không đệ quy ép buộc tính không kết hợp."

Left-recursive rules (Quy tắc đệ quy trái): Quy tắc có dạng: nonTerminal ::= nonTerminal operator ... (nonTerminal xuất hiện đầu tiên ở vế phải).
enforce left-associativity (ép buộc tính kết hợp từ trái sang phải): Khi sử dụng left recursion, các phép toán cùng loại sẽ được nhóm lại và thực hiện từ trái sang phải.
expr ::= expr '-' term | term // Left recursion ở `expr '-' term`
term ::= ...
Quy tắc expr ::= expr '-' term là left-recursive vì expr xuất hiện đầu tiên ở vế phải. Grammar này sẽ làm cho phép trừ - có tính kết hợp từ trái sang phải (ví dụ: a - b - c hiểu là (a - b) - c).

Right-recursive rules enforce right-associativity.

Tạm dịch: "• Quy tắc đệ quy phải ép buộc tính kết hợp từ phải sang trái."

Right-recursive rules (Quy tắc đệ quy phải): Quy tắc có dạng: nonTerminal ::= ... operator nonTerminal (nonTerminal xuất hiện ở cuối vế phải).

enforce right-associativity (ép buộc tính kết hợp từ phải sang trái): Khi sử dụng right recursion, các phép toán cùng loại sẽ được nhóm lại và thực hiện từ phải sang trái.

Ví dụ (phép gán right-associative):
assignment ::= ID '=' assignment | expression // Right recursion ở `ID '=' assignment`
expression ::= ...
Quy tắc assignment ::= ID '=' assignment là right-recursive vì assignment xuất hiện ở cuối vế phải. Grammar này có thể làm cho phép gán = có tính kết hợp từ phải sang trái (ví dụ: a = b = 5 hiểu là a = (b = 5)). (Lưu ý: tính kết hợp của phép gán có thể phức tạp và khác nhau tùy ngôn ngữ).

Non-recursive rules enforce non-associativity.

Tạm dịch: "• Quy tắc không đệ quy ép buộc tính không kết hợp."

Non-recursive rules (Quy tắc không đệ quy): Quy tắc không chứa đệ quy trực tiếp (non-terminal ở vế trái không xuất hiện lại ở vế phải của cùng quy tắc đó).

enforce non-associativity (ép buộc tính không kết hợp): Nếu bạn muốn toán tử là không kết hợp (non-associative) (nghĩa là, biểu thức như a op b op c là không hợp lệ, hoặc cần dấu ngoặc đơn để làm rõ thứ tự), bạn có thể sử dụng quy tắc non-recursive.
comparison ::= expression RELOP expression // Non-recursive rule

Formula 5: An infix expression

Quy tắc comparison ::= expression RELOP expression là non-recursive vì comparison không xuất hiện ở vế phải. Grammar này có thể làm cho toán tử quan hệ RELOP (ví dụ: <) trở thành non-associative. Ví dụ, a < b < c có thể không được phép hoặc cần được viết rõ ràng hơn như (a < b) < c (nếu logic của ngôn ngữ cho phép).

1. **Operator Precedence:** The closer an operator's production rule is to the start production rule, the lower its precedence.
2. **Operator Associativity**:
   - **Left-recursive** rules enforce **left-associativity**.
   - **Right-recursive** rules enforce **right-associativity**.
   - **Non-recursive** rules enforce **non-associativity**.

Ghi nhớ

Gần gốc ưu tiên thấp hơn,
Trái thì kết trái, phải thì cũng xuôi.
Không đệ quy đứng lặng thôi,
Giữa dòng toán tử, một nơi vững vàng.

MEng. Tran Ngoc Bao Duy

THANK YOU.