



HK2 2024-2025

Principles of Programming Languages

Vu Van Tien

Review

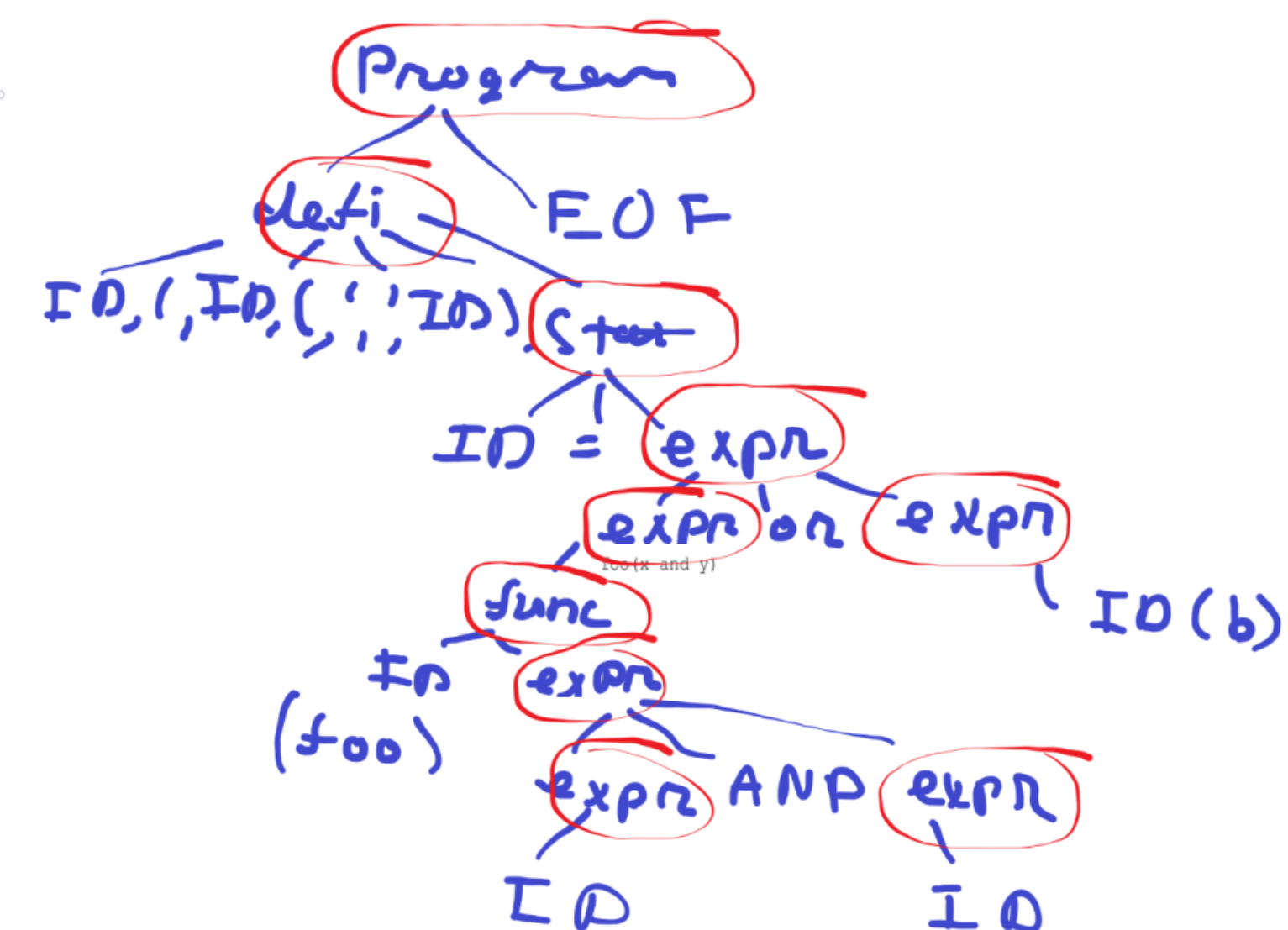
Review Midterm

Questions: 9, 11, 12, 13, 18, 25, 34, 36, 41, 42

Áp dụng mô tả sau cho các câu 6–9:

Cho ngôn ngữ X được mô tả trong ANTLR4 như sau:

```
1 grammar X;
2 program: stat EOF | defi EOF;
3 stat: ID '=' expr ';' | expr ';';
4 defi : ID '(' ID (',' ID)* ')' '{' stat* '}' ;
5 expr: ID | INT | func | 'not' expr | expr 'and' expr | expr 'or' expr;
6 func : ID '(' expr (',' expr)* ')' ;
7 INT : [0-9]+ ;
8 ID: [a-zA-Z_][a-zA-Z_0-9]* ;
9 WS: [ \t\n\r\f]+ -> skip ;
```



Câu 9. [A1-2] Với chuỗi nhập $f(x, y) \{ a = \text{foo}(x \text{ and } y) \text{ or } b; \}$, cây phân tích cú pháp sinh ra bởi văn phạm trên có bao nhiêu nút ứng với ký hiệu không kết thúc?

(A) 10

(B) 9

(C) 8



(D) 7

Review Midterm

Questions: 9, 11, 12, 13, 18, 25, 34, 36, 41, 42

Biểu thức chính quy không thể "đếm" hoặc "nhớ" một số lượng không giới hạn các ký tự để so sánh với một phần khác của chuỗi.

Câu 25. Ngôn ngữ nào sau đây có thể mô tả bằng một biểu thức chính quy?

- (A) $L = \{ 0^n 1^n \mid n \geq 0 \}$
- (B) $L = \{ x \mid x \text{ chứa số lượng } 0 \text{ bằng số lượng } 1 \}$
-  (C) $L = \{ x \mid x \text{ chứa số lượng } 0 \text{ chẵn} \}$ 
- (D) $L = \{ x \mid x \text{ là một chuỗi đối xứng (palindrome) trên } \{0,1\} \}$

(Nó là automata -> Không thể đếm và nhớ)

- **A. $L = \{ 0^n 1^n \mid n \geq 0 \}$:** Ngôn ngữ này yêu cầu số lượng ký tự '0' phải bằng số lượng ký tự '1'. Điều này đòi hỏi khả năng đếm số lượng '0' và sau đó khớp với cùng số lượng '1'. Biểu thức chính quy **không thể** làm điều này vì n có thể lớn tùy ý. Đây là một ví dụ kinh điển của ngôn ngữ phi ngữ cảnh (context-free language) không phải là ngôn ngữ chính quy.
- **B. $L = \{ x \mid x \text{ chứa số lượng } 0 \text{ bằng số lượng } 1 \}$:** Tương tự như câu A, việc đảm bảo số lượng '0' bằng số lượng '1' trong một chuỗi bất kỳ (không nhất thiết phải theo thứ tự 0 rồi đến 1) cũng đòi hỏi khả năng "đếm" và "ghi nhớ" không giới hạn. Biểu thức chính quy **không thể** mô tả ngôn ngữ này.
- **C. $L = \{ x \mid x \text{ chứa số lượng } 0 \text{ chẵn} \}$:** Ngôn ngữ này có thể được mô tả bằng biểu thức chính quy. Chúng ta có thể sử dụng một máy tự động hữu hạn với hai trạng thái: một trạng thái biểu thị số lượng '0' đã gặp là chẵn, và trạng thái kia biểu thị số lượng '0' đã gặp là lẻ.
 - Trạng thái ban đầu là "chẵn".
 - Nếu đang ở trạng thái "chẵn" và gặp '0', chuyển sang trạng thái "lẻ".
 - Nếu đang ở trạng thái "lẻ" và gặp '0', chuyển sang trạng thái "chẵn".
 - Gặp '1' không làm thay đổi trạng thái chẵn/lẻ của số lượng '0'.
 - Trạng thái chấp nhận là trạng thái "chẵn".Một biểu thức chính quy ví dụ có thể là: $1^*(01^*01^*)^*1^*$ (hoặc các biến thể phức tạp hơn để bao gồm tất cả các trường hợp). Về cơ bản, ý tưởng là sau mỗi cặp số 0, bạn có thể có bất kỳ số lượng số 1 nào.
- **D. $L = \{ x \mid x \text{ là một chuỗi đối xứng (palindrome) trên } \{0,1\} \}$:** Palindrome yêu cầu ký tự đầu tiên phải giống ký tự cuối cùng, ký tự thứ hai giống ký tự áp chót, và cứ thế. Điều này đòi hỏi việc "nhớ" nửa đầu của chuỗi để so sánh với nửa sau theo thứ tự ngược lại. Với độ dài chuỗi không giới hạn, biểu thức chính quy **không thể** làm điều này. Đây cũng là một ví dụ của ngôn ngữ phi ngữ cảnh không chính quy.

Vậy, câu trả lời đúng là C. Ngôn ngữ mô tả các chuỗi có số lượng ký tự '0' chẵn có thể được mô tả bằng một biểu thức chính quy.

Review Midterm

Questions: 9, 11, 12, 13, 18, 25, 34, 36, 41, 42

Câu 36. Cho một văn phạm được viết trên ANTLR4 như sau:

```
grammar Expr;
```

```
expr: expr ('+' | '-') term | expr ('*' | '/') term | term;  
term: '(' expr ')' | INT;
```

```
INT : [0-9]+ ;
```

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

và các lớp AST được định nghĩa trong Python:

```
class AST: pass
```

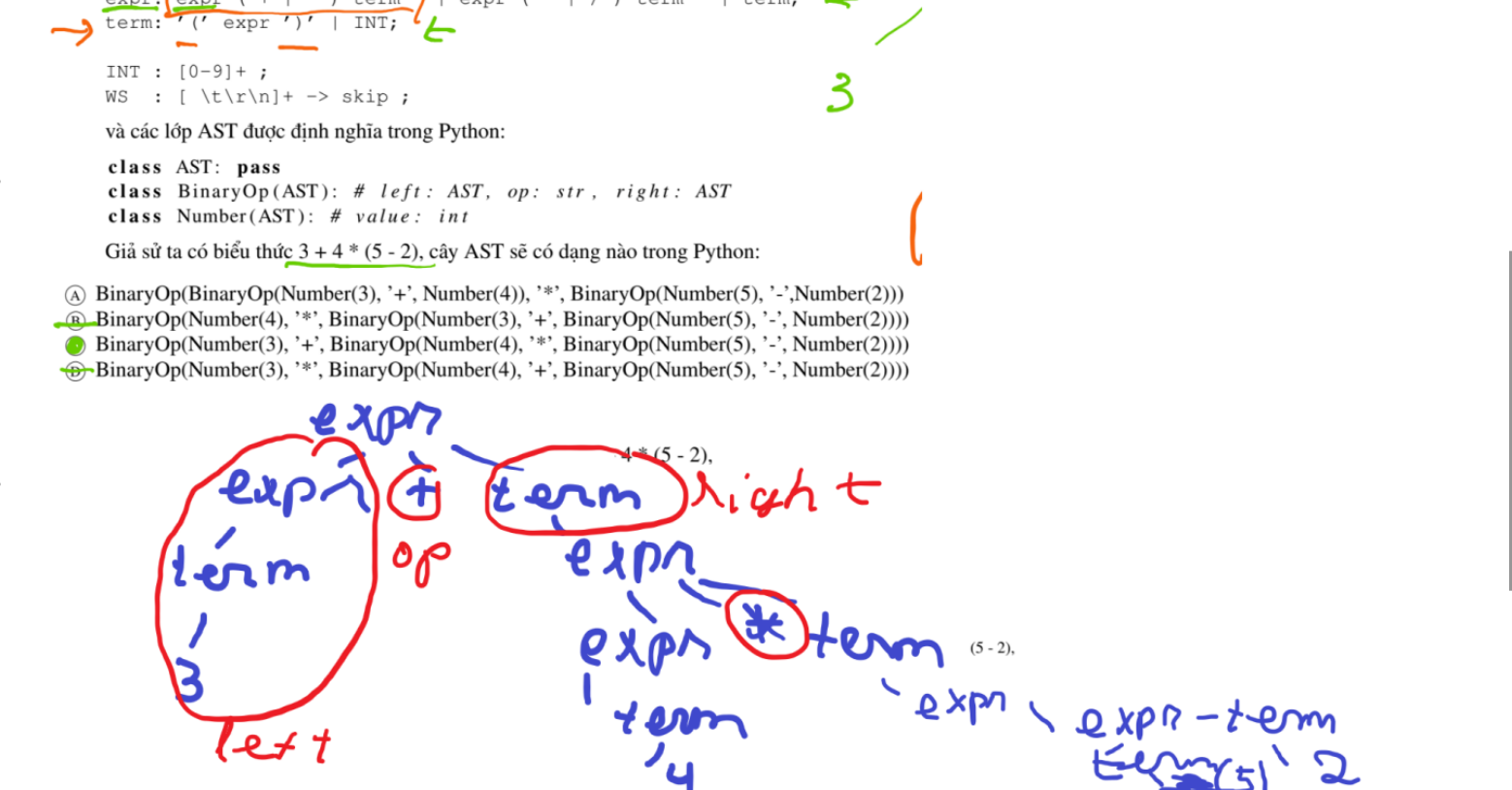
```
class BinaryOp(AST): # left: AST, op: str, right: AST
```

```
class Number(AST): # value: int
```

Giả sử ta có biểu thức $3 + 4 * (5 - 2)$, cây AST sẽ có dạng nào trong Python:

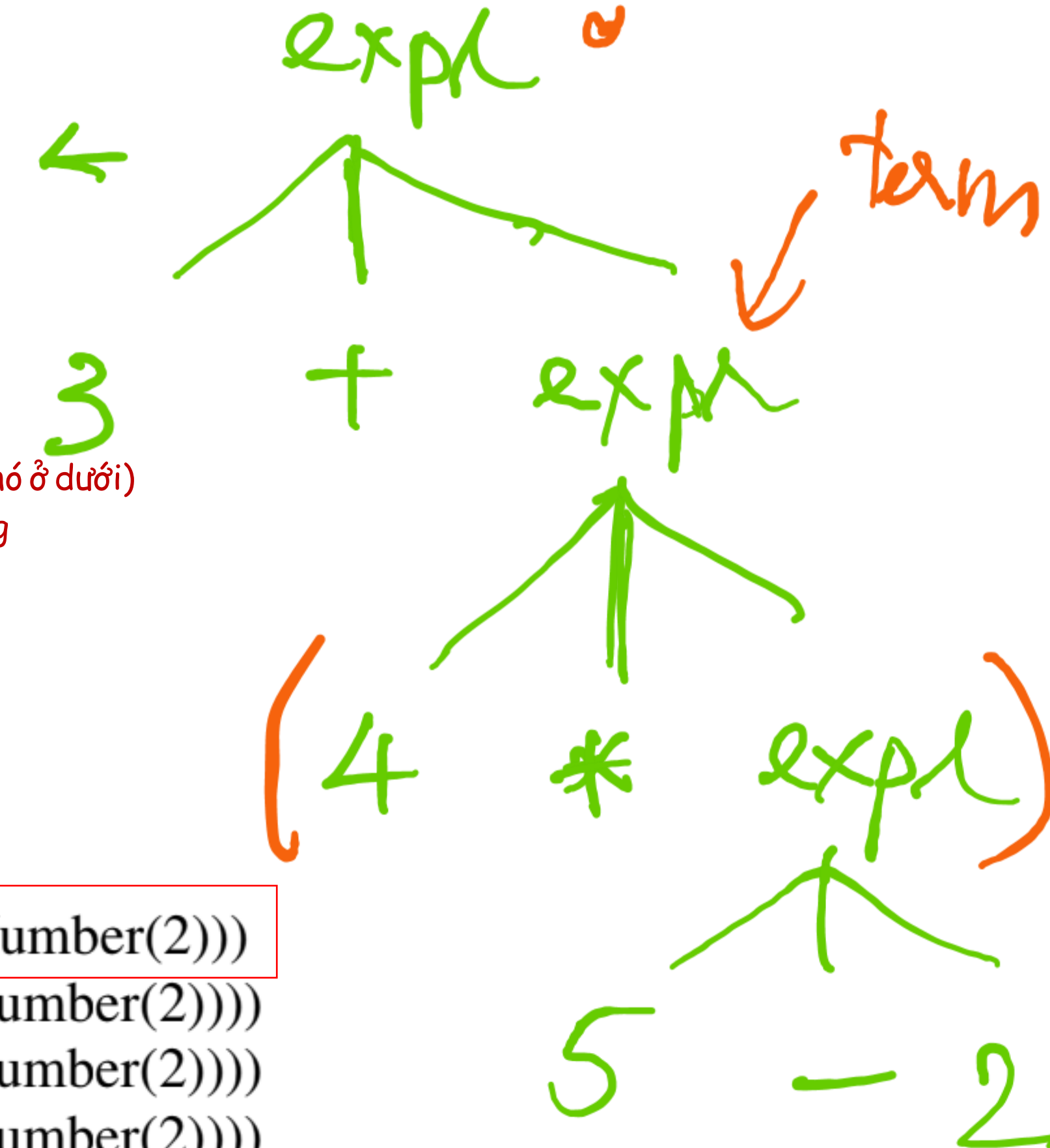
- ☐ A BinaryOp(BinaryOp(Number(3), '+', Number(4)), '*', BinaryOp(Number(5), '-', Number(2)))
- ☒ B BinaryOp(Number(4), '*', BinaryOp(Number(3), '+', BinaryOp(Number(5), '-', Number(2))))
- ☐ C BinaryOp(Number(3), '+', BinaryOp(Number(4), '*', BinaryOp(Number(5), '-', Number(2))))
- ☐ D BinaryOp(Number(3), '*', BinaryOp(Number(4), '+', BinaryOp(Number(5), '-', Number(2))))

- kết hợp trái
- độ ưu tiên



Xử lý term trước -> $(5-2)$
Sau đó xử lý lên trên, gộp $3 + 4$ -> khớp
-> $(3+4)*(5-2)$

Nghĩa là term có độ ưu tiên lớn hơn expr (vì nó ở dưới)
Sau đó đi từ trái sang phải như bình thường



Thứ tự "gọi" (hay dẫn xuất):

Để có một *expr*, bạn có thể bắt đầu bằng một *term*.

Để có một *term*, bạn có thể bắt đầu bằng một *factor*.

Một *factor* có thể là một số (INT) hoặc một biểu thức nằm trong ngoặc (*' expr '*).

Quá trình "khớp" của parser (tưởng tượng theo hướng bottom-up hoặc khi xây dựng cây):

Parser sẽ cố gắng nhận diện các đơn vị nhỏ nhất trước, đó là các *factor* (ví dụ: số nguyên hoặc biểu thức trong ngoặc).

Sau đó, nó sẽ cố gắng kết hợp các *factor* này (và các *term* khác) để tạo thành các *term* bằng cách sử dụng các toán tử có độ ưu tiên cao hơn (ví dụ: *).

Cuối cùng, nó sẽ cố gắng kết hợp các *term* này (và các *expr* khác) để tạo thành các *expr* bằng cách sử dụng các toán tử có độ ưu tiên thấp hơn (ví dụ: +).

Trong trường hợp văn phạm bạn đưa ra ban đầu:

term có "độ ưu tiên" cao hơn *expr* trong việc được nhận diện như một khối đơn lẻ. Nghĩa là, một chuỗi con nào đó nếu có thể được hiểu là một *term* (ví dụ, một INT hoặc một (*expr*)) thì nó sẽ được "gom" lại thành *term* trước.

Khi bạn có (5 - 2):

Bên trong ngoặc () là một *expr* (5 - 2).

Toàn bộ cụm (5 - 2) khớp với luật *term* : *'(' expr ')'*. Do đó, nó được coi là một *term*.

Đối với các toán tử + và *:

Trong văn phạm này, cả hai đều nằm trong các luật của *expr* (*expr* : *expr OP term*).

Vì (5 - 2) đã được "đóng gói" thành một *term* duy nhất (gọi là *T_ngoac*), nên biểu thức trở thành 3 + 4 * *T_ngoac*.

Do tính đệ quy trái của các luật *expr*, parser sẽ xử lý từ trái sang phải:

3 + 4 sẽ được nhóm trước thành một *expr* (vì 3 là *term*, + là toán tử, 4 là *term*).

Sau đó, kết quả của (3 + 4) mới được nhân với *T_ngoac*.

Biểu thức cần phân tích: 3 + 4 * (5 - 2)

Quá trình phân tích cú pháp (Parsing) và xây dựng cây AST (theo hướng top-down, leftmost derivation - một cách phổ biến mà parser có thể hoạt động):

Mục tiêu của parser là tìm cách dẫn xuất chuỗi token đầu vào từ ký hiệu bắt đầu của văn phạm (trong trường hợp này là *expr*). Cây AST sẽ được hình thành dựa trên cấu trúc của các luật được áp dụng.

Bắt đầu với *expr* (ký hiệu bắt đầu):

Biểu thức của chúng ta là 3 + 4 * (5 - 2). Parser sẽ cố gắng khớp biểu thức này với một trong các luật của *expr*.

Áp dụng Luật *expr*:

Parser sẽ thử các luật của *expr* từ trên xuống dưới hoặc theo một chiến lược nào đó. Do tính đệ quy trái trong *expr* : *expr* ... term, parser thường sẽ cố gắng khớp phần term trước, rồi mới xem xét các toán tử.

Tuy nhiên, một điểm quan trọng hơn trong văn phạm này là cách term được định nghĩa:

term : '(' *expr* ')' | INT;

Điều này có nghĩa là một term có thể là:

Một biểu thức nằm trong dấu ngoặc đơn ('(' *expr* ')').

Hoặc một số nguyên (INT).

Khi parser "nhìn" vào chuỗi 3 + 4 * (5 - 2), nó sẽ thấy cụm (5 - 2). Cụm này khớp hoàn hảo với luật term : '(' *expr* ')'.

Xử lý (5 - 2) như một term:

Parser nhận diện (là ký hiệu mở đầu của một term (theo luật term1).

Bên trong dấu ngoặc là 5 - 2. Parser sẽ đệ quy gọi chính nó để phân tích 5 - 2 như một *expr* (gọi là *expr_trong_ngoac*).

Để phân tích *expr_trong_ngoac* (5 - 2), parser sẽ áp dụng luật *expr* : *expr* ('+' | '-') term (luật *expr1*).

expr bên trái của - là 5. 5 sẽ được phân tích thành term (theo luật *expr3*), rồi term này sẽ được phân tích thành INT (theo luật *term2*). AST: Number(5).

Toán tử là '-'.

term bên phải của - là 2. 2 sẽ được phân tích thành INT (theo luật *term2*). AST: Number(2).

Vậy, AST cho *expr_trong_ngoac* (5 - 2) là: BinaryOp(Number(5), '-', Number(2)).

Parser nhận diện) là ký hiệu kết thúc của term (theo luật *term1*).

Như vậy, toàn bộ cụm (5 - 2) được xem như một term đơn lẻ trong biểu thức lớn hơn, và AST của nó là BinaryOp(Number(5), '-', Number(2)). (Gọi AST này là *AST_ngoac*)

Biểu thức còn lại sau khi xử lý phần trong ngoặc:

Sau khi (5 - 2) được xử lý và có AST là *AST_ngoac*, biểu thức của chúng ta có thể được xem như là 3 + 4 * *AST_ngoac*.

Tiếp tục phân tích *expr* cho 3 + 4 * *AST_ngoac*:

Bây giờ, parser sẽ áp dụng các luật *expr* cho biểu thức này. Do văn phạm *expr* có đệ quy trái và không phân tách rõ ràng độ ưu tiên giữa + và * ở cùng một cấp (cả hai đều là *expr* ... term), nó sẽ có xu hướng nhóm từ trái sang phải.

3 + 4 sẽ được nhóm trước:

Parser thấy 3 là một term (-> INT).

Sau đó là toán tử +.

Tiếp theo là 4, cũng là một term (-> INT).

Luật *expr* : *expr* ('+' | '-') term được áp dụng.

AST cho 3 + 4 là: BinaryOp(Number(3), '+', Number(4)) (Gọi AST này là *AST_cong*)

Kết hợp với phần còn lại:

Bây giờ biểu thức trở thành *AST_cong* * *AST_ngoac*.

Parser áp dụng luật *expr* : *expr* ('*' | '/') term.

expr bên trái là *AST_cong*.

Toán tử là *.

term bên phải là *AST_ngoac*.

AST cuối cùng sẽ là: BinaryOp(*AST_cong*, '*', *AST_ngoac*)

=> BinaryOp(BinaryOp(Number(3), '+', Number(4)), '*', BinaryOp(Number(5), '-', Number(2)))

Tại sao dấu ngoặc lại được ưu tiên?

Trong hầu hết các văn phạm (bao gồm cả văn phạm này), dấu ngoặc đơn () được dùng để thay đổi hoặc ép buộc thứ tự ưu tiên tự nhiên của các toán tử. Luật term : '(' *expr* ')' đảm bảo rằng bất cứ thứ gì nằm bên trong dấu ngoặc đơn sẽ được coi như một đơn vị *expr* hoàn chỉnh và được lượng giá (hoặc xây dựng cây AST cho nó) trước khi kết hợp với các toán tử bên ngoài dấu ngoặc.

Nói cách khác, khi parser gặp (, nó biết rằng nó phải phân tích toàn bộ biểu thức bên trong cho đến khi gặp) tương ứng, và coi kết quả của biểu thức bên trong đó như một term đơn lẻ. Sau đó, term này mới được sử dụng trong các phép toán ở cấp độ cao hơn.

Tóm lại:

Dấu ngoặc đơn () trong luật term : '(' *expr* ')' có độ ưu tiên cao nhất, buộc parser phải xử lý biểu thức bên trong ngoặc trước.

Sau khi phần trong ngoặc (5 - 2) được xử lý thành một đơn vị (có AST riêng), văn phạm *expr* với các luật đệ quy trái sẽ khiến các phép toán + và * được nhóm từ trái sang phải (trong trường hợp này, + trước rồi đến *).

Đây là lý do tại sao đáp án A là kết quả của việc phân tích cú pháp chặt chẽ theo văn phạm đã cho, mặc dù nó không phản ánh độ ưu tiên toán học thông thường.

Review Midterm

- kết hợp trái
- độ ưu tiên

Questions: 9, 11, 12, 13, 18, 25, 34, 36, 41, 42

Câu 36. Cho một văn phạm được viết trên ANTLR4 như sau:

```
grammar Expr;
```

```
expr: expr ('+' | '-') term | expr ('*' | '/') term | term;
```

```
term: '(' expr ')' | INT;
```

```
INT : [0-9]+ ;
```

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

và các lớp AST được định nghĩa trong Python:

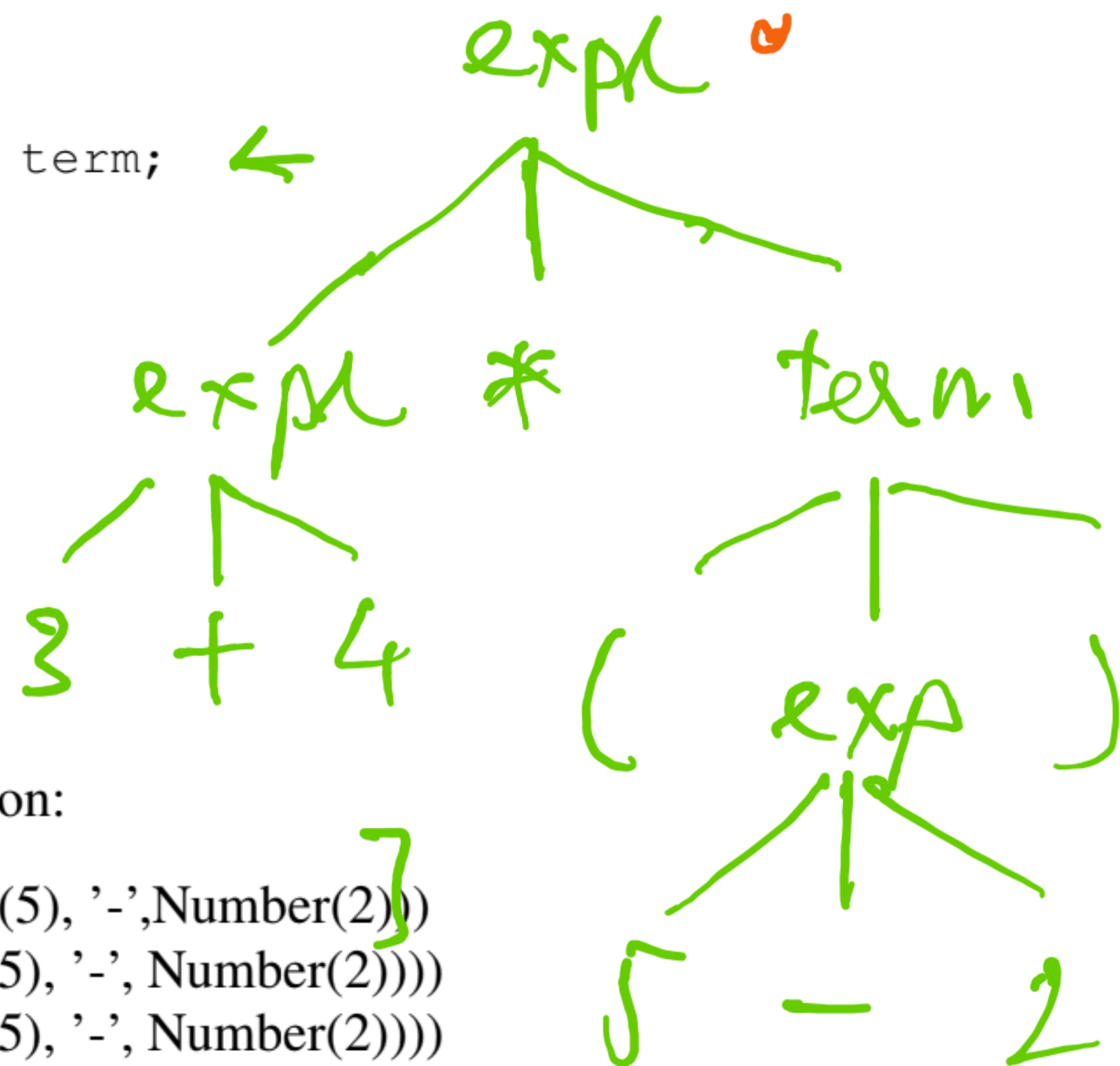
```
class AST: pass
```

```
class BinaryOp(AST): # left: AST, op: str, right: AST
```

```
class Number(AST): # value: int
```

Giả sử ta có biểu thức $3 + 4 * (5 - 2)$, cây AST sẽ có dạng nào trong Python:

- ☐ A BinaryOp(BinaryOp(Number(3), '+', Number(4)), '*', BinaryOp(Number(5), '-', Number(2)))
- ☐ B BinaryOp(Number(4), '*', BinaryOp(Number(3), '+', BinaryOp(Number(5), '-', Number(2))))
- ☒ C BinaryOp(Number(3), '+', BinaryOp(Number(4), '*', BinaryOp(Number(5), '-', Number(2))))
- ☐ D BinaryOp(Number(3), '*', BinaryOp(Number(4), '+', BinaryOp(Number(5), '-', Number(2))))



Review Midterm

Questions: 9, 11, 12, 13, 18, 25, 34, 36, 41, 42

Câu 41. [L.O.3.1] Kết quả của `E().show()` dựa vào đoạn mã Python sau là gì?

```
class A:
    def show(self):
        print("A", end="")
```

```
class B(A):
    def show(self):
        super().show()
        print("B", end="")
```

```
class C(A):
    def show(self):
        print("C", end="")
        super().show()
```

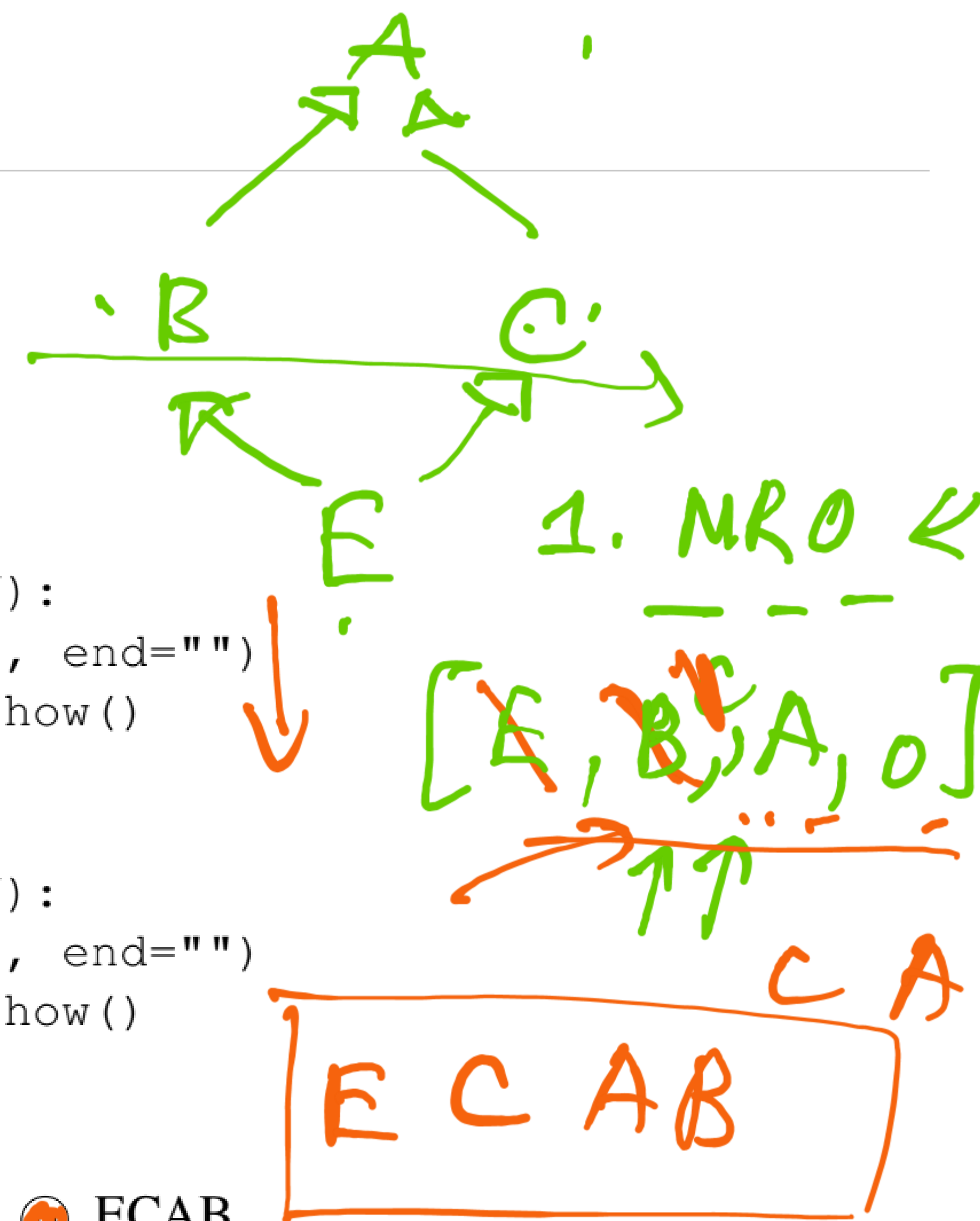
```
class E(B,C):
    def show(self):
        print("E", end="")
        super().show()
```

(A) EABCA

(B) EBCA

(C) EAB

☒ (D) ECAB



$MRO[A] = [A, Object]$

$MRO[B] = [B] + MRO[A] = [B, A, Object]$

$MRO[C] = [C] + MRO[A] = [C, A, Object]$

$MRO[E] = [E] + MRO[B] + MRO[C] = [E, B, C, A, OBJECT]$

$E().show() \rightarrow Print(E) \rightarrow E$

Gọi $super().show()$ \rightarrow Tìm $show()$ trong class tiếp theo của $MRO(E) \rightarrow B$ được gọi \rightarrow gọi $B.show()$

Khi $B.show()$ được gọi \rightarrow sẽ gọi $super().show()$ của B (1), sau đó $print(B)$ (2)

(1): Tìm lớp tiếp theo trong MRO của $(B) \rightarrow$ Nhìn vào $MRO(E) \rightarrow$ Sau B là C , khi đó khi gọi $super().show()$ thì $show()$ của C sẽ được gọi $\rightarrow In C \Rightarrow EC$.

Sau đó, $super().show()$ của C lại được gọi, nhìn vào $MRO(E) \rightarrow$ Sau C là $A \rightarrow$ Gọi $show()$ của $A \Rightarrow In A \Rightarrow ECA$

Kết quả sau (1): ECA

(2) $\Rightarrow ECAB$.

Review Midterm

Đoạn mô tả sau áp dụng cho các câu hỏi 14–18:

Cho văn phạm viết trên ANTLR4 mô tả các khai báo thông số trong một khai báo hàm của MiniGo như sau:

```
params: param (CM param)*;  
param: ids mtype ; // mtype represents a type  
ids: ID CM ids | ID ;
```

Một param bao gồm một danh sách các ids (tên biến) và một mtype (kiểu của chúng).

cùng với lớp ParamDecl trên AST để biểu diễn cho một khai báo thông số như sau:

```
@dataclass  
class ParamDecl(Decl):  
    parName: str # name of one parameter  
    parType: Type
```

Mỗi đối tượng ParamDecl đại diện cho một tham số, bao gồm tên (parName) và kiểu (parType).

và đoạn mã sau để sinh ra AST cho các khai báo thông số như một danh sách các đối tượng ParamDecl:

```
1 def visitParams(self, ctx: MiniGoParser.ParamsContext):  
2     return _____(1)_____  
3 def visitParam(self, ctx: MiniGoParser.ParamContext):  
4     a = ctx.getChild(0).accept(self)  
5     b = ctx.getChild(1).accept(self)  
6     return [_____(2)_____]_____  
7 def visitIds(self, ctx: MiniGoParser.IdsContext):  
8     id = ctx.ID().getText()  
9     if ____ (3) ____ == 3: return [id] + ____ (4) ____  
10    return __ (5) ____  
11 def visitMtype(self, ctx: MiniGoParser.MtypeContext): # return Type
```

Mục tiêu của visitParams là trả về một danh sách các đối tượng ParamDecl.

tạo ra một danh sách các đối tượng ParamDecl từ cây cú pháp (parse tree).

a sẽ là kết quả của việc visit nút ids (là con đầu tiên của param). Dựa vào hàm visitIds, a sẽ là một danh sách các chuỗi (tên biến),
b sẽ là kết quả của việc visit nút mtype (là con thứ hai của param). Dựa vào hàm visitMtype (dòng 11), b sẽ là một đối tượng Type.

Hàm visitParam cần trả về một danh sách các đối tượng ParamDecl. Vì a là một danh sách các tên và b là một kiểu chung cho tất cả các tên đó, nên chỗ trống (2) sẽ tạo ra các đối tượng ParamDecl(name, b) cho mỗi name trong a.

A:

Review Midterm

y ở đây là một ParamContext.

y.accept(self) (tương đương self.visitParam(y)) sẽ trả về một danh sách các ParamDecl.

x + danh_sach_ParamDecl sẽ nối danh sách này vào x (với x khởi tạo là []).

Đây là một lựa chọn khả thi. Nó sử dụng reduce để tích lũy kết quả từ việc visit mỗi param.

Câu 18. [A2] Hãy chọn mã thích hợp cho chỗ trống (1):

- ☒ `functools.reduce(lambda x,y: x+y.accept(self),ctx.param(),[])`
- ☐ `functools.reduce(lambda x,y:x.accept(self)+y,ctx.param(),[])`
- ☐ `list(map(lambda x:self.visit(x),ctx.param()))`
- ☐ `list(map(lambda x:self.visit(x),ctx.param()))`

[✓ Decl]

[accept
visit...y]

[]

→ reduce (lambda acc, ele : x + [], List, initial)

→ list (map (

→ filter : tìm sym : name == id.name

b: x.accept(self): x là accumulator (ban đầu là []). [].accept(self) không hợp lệ. Loại.

c:) list(map(lambda x:self.visit(x),ctx.param()))

map sẽ áp dụng self.visit (chính là self.visitParam) cho mỗi param trong ctx.param().

Mỗi lần gọi self.visitParam trả về một danh sách các ParamDecl.

Kết quả của map sẽ là một danh sách các danh sách các ParamDecl. Ví dụ: [[ParamDecl1, ParamDecl2], [ParamDecl3], ...].

Hàm list() chỉ chuyển iterator của map thành list, không làm phẳng nó.

Kết quả không phải là một danh sách phẳng các ParamDecl. Loại.

d: list(map(lambda x:self.visit(x),ctx.param()))

self.visit(x) trả về một danh sách. Danh sách không có thuộc tính ctx. Lỗi cú pháp. Loại.

```
def visitParams(self, ctx:MiniGoParser.ParamsContext):
    return _____(1)_____
```

ctx ở đây là một ParamsContext, tương ứng với rule params: param (CM param)*:.

Nghĩa là ctx chứa một danh sách các nút param. Chúng ta có thể truy cập chúng thông qua ctx.param(). Lưu ý rằng ctx.param() sẽ trả về một danh sách các ParamContext.

Mục tiêu của visitParams là trả về một danh sách các đối tượng ParamDecl.

Hàm visitParam đã được thiết kế để xử lý một nút param và trả về một danh sách các ParamDecl (vì một param có thể khai báo nhiều ids cùng một mtype).

Xây dựng logic cho chỗ trống (1):

Chúng ta cần duyệt qua tất cả các nút param con của ctx, gọi self.visit(param_node) (hoặc param_node.accept(self)) cho mỗi nút đó, và kết hợp các kết quả lại. Vì visitParam trả về một danh sách các ParamDecl, và visitParams cũng cần trả về một danh sách các ParamDecl (tổng hợp từ tất cả các param), chúng ta cần một cách để "làm phẳng" (flatten) danh sách của các danh sách.