ÔN THI GIỮA KỲ
Môn học: Nguyên lý ngôn ngữ lập trình

Chương 1: GIỚI THIỆU

Tại sao chương trình nhanh hơn sau lặp lại: Hiện tượng chương trình chạy chậm lúc đầu và nhanh hơn sau lặp lại là đặc trưng của JIT. Lần đầu tiên các đoạn mã (đặc biệt là các vòng lặp) được thực thi, chúng có thể được thông dịch. Nhưng khi vòng lặp chạy lại, JIT đã có thể biên dịch vòng lặp đó thành mã máy, giúp các lần lặp sau chạy nhanh hơn nhiều.

Câu 01. Một lập trình viên viết chương trình bằng ngôn ngữ Z. Khi chạy, anh ta nhận thấy chương trình bắt đầu chậm nhưng dần trở nên nhanh hơn sau vài lần lặp lại mà không cần biên dịch lại. Dựa trên hiện tượng này, hãy xác định phương pháp dịch có thể đã được sử dụng và giải thích cơ chế hoạt động của nó.

Câu 02. Một trình dịch lai mất 3 giây để biên dịch mã nguồn thành bytecode, sau đó thực thi với tốc độ 0,8 giây mỗi dòng lệnh. Trong khi đó, một trình thông dịch thuần túy mất 2,5 giây để thực thi mỗi dòng lệnh. Hãy xác định số dòng lệnh tối thiểu để trình dịch lai có tổng thời gian nhỏ hơn trình thông dịch thuần túy.

3 + 0.8n < 2.5n

Câu 03. Một chương trình có hai lựa chọn để thực thi:

Biên dịch: 7 + 0.5n

- Dùng biên dịch truyền thống: Mất 7 giây để biên dịch, nhưng sau đó chạy với tốc độ 0,5 giây mỗi dòng lệnh.
- Dùng trình dịch tức thời (JIT): Không mất thời gian biên dịch ban đầu, nhưng mỗi dòng mất 1,2 giây khi chạy lần đầu và 0,4 giây khi chạy lại.

0.4n * 0.4 + 0.6n * 1.2

Với chương trình có N dòng lệnh, trong đó 40% số dòng được thực thi 2 lần, hãy tìm giá trị N tối thiểu để trình dịch tức thời (JIT) có tổng thời gian thực thi nhanh hơn phương pháp biên dịch truyền thống.

Câu 04. Công cụ nào có nhiệm vụ kết hợp các tệp đối tượng (object files) lại với nhau và tạo ra một tệp thực thi?

Khi trình biên dịch (compiler) biên dịch mã nguồn (ví dụ, file .c, .cpp), nó không tạo ra trực tiếp tệp thực thi hoàn chỉnh. Thay vào đó, nó tạo ra các tệp đối tượng (.o hoặc .obj).

Câu 05. Điền vào chỗ trống: "_____ đảm bảo rằng tất cả các chương trình và thư viện cần thiết được nạp vào bộ nhớ trước khi thực thi."

Trình nạp là một phần của hệ điều hành, có nhiệm vụ nạp chương trình thực thi và các thư viện cần thiết vào bộ nhớ (RAM) để chuẩn bị cho việc thực thi.

Chương 2: PHÂN TÍCH TỪ VỰNG

Câu 01. Cho các loại lỗi sau:

- Một biến chưa được khai báo nhưng được sử dụng
- Một từ khóa được sử dụng làm tên biến
- Một dấu @ xuất hiện trong mã nguồn C
- Một dấu ngoặc đóng bị thiếu
- Một số nguyên có dấu chấm thập phân.

Có bao nhiêu lỗi trong danh sách trên mà bộ phân tích từ vựng có thể phát hiện?

Câu 02. Cho đoạn mã sau được viết bằng C++:

int 1var = 10; float num = 12.; char ch = '$';

Có bao nhiêu lỗi mà bộ phân tích từ vựng (lexical analyzer) có thể phát hiện trong đoạn mã trên?

Câu 03. Trong một tập luật của ANTLR4, bạn có quy tắc lexer như sau: `STRING : '"' .*? '"';`. Quy tắc này cho phép nhận diện chuỗi nằm giữa hai dấu " nhưng không thay đổi nội dung bên trong. Bạn muốn thay đổi action để mọi chuỗi "hello" được thay bằng "STRING CONSTANT" khi token được xử lý. Hãy mô tả cách thực hiện điều này và giải thích cách thức hoạt động của lexer trong ANTLR4 để đạt được kết quả mong muốn.

```
STRING
  : '"' text=.*? '"'  -> { self.text = "STRING CONSTANT" if self.text == "hello" else self.text }
  ;
```

Ví dụ tham lam - không tham lam:
<a>text1 <a>text2

1. Mẫu tham lam: <a>.*

<a>: Khớp với dấu mở thẻ <a>.
.*: Tham lam - Khớp với bất kỳ ký tự nào (.), không hoặc nhiều lần (*), một cách tham lam. Phần này sẽ cố gắng khớp với càng nhiều ký tự càng tốt.
: Khớp với dấu đóng thẻ .
Cách mẫu tham lam hoạt động:

Mẫu <a> khớp với <a> đầu tiên.
.* (tham lam) bắt đầu khớp từ ký tự t trong "text1". Nó sẽ tiếp tục khớp với tất cả các ký tự cho đến khi nó gặp dấu cuối cùng trong chuỗi input. Điều này là vì nó muốn khớp càng nhiều càng tốt.
 khớp với cuối cùng.
Kết quả khớp tham lam: Mẫu <a>.* sẽ khớp với toàn bộ chuỗi: <a>text1 <a>text2. Nó đã "ăn" luôn cả phần <a>text2 vì định lượng .* là tham lam và cố gắng khớp càng nhiều càng tốt trước khi nhả ra để phần còn lại của mẫu () có thể khớp.

2. Mẫu không tham lam: <a>.*?

<a>**: Khớp với dấu mở thẻ <a>.
.*?: Không tham lam - Khớp với bất kỳ ký tự nào (.), không hoặc nhiều lần (*), một cách không tham lam. Phần này sẽ cố gắng khớp với càng ít ký tự càng tốt.
: Khớp với dấu đóng thẻ .
Cách mẫu không tham lam hoạt động:

Mẫu <a> khớp với <a> đầu tiên.
.*? (không tham lam) bắt đầu khớp từ ký tự t trong "text1". Nó sẽ cố gắng khớp với ít ký tự nhất có thể. Nó sẽ dừng lại ngay khi có thể để phần còn lại của mẫu () có thể khớp. Trong trường hợp này, nó sẽ dừng lại ngay trước gần nhất.
 khớp với gần nhất.
Kết quả khớp không tham lam: Mẫu <a>.*? sẽ khớp với khớp đầu tiên: <a>text1. Sau đó, khi tiếp tục tìm kiếm, nó sẽ tìm thấy khớp thứ hai: <a>text2. Mẫu không tham lam sẽ khớp hai lần, mỗi lần khớp với một cặp thẻ <a>... riêng biệt.

Quy tắc .*? là non-greedy nhưng vẫn có thể gây lỗi với chú thích lồng nhau: Quy tắc .*? (khớp bất kỳ ký tự nào, không hoặc nhiều lần, một cách không tham lam) được thiết kế để khớp với nội dung ngắn nhất có thể giữa /* và */. Tuy nhiên, nó không xử lý đúng trường hợp chú thích đa dòng lồng nhau.

Câu 04. Một bộ phân tích từ vựng sử dụng quy tắc lexer như sau:

```
MULTI_LINE_COMMENT : '/*' .*? '*/' -> skip;
SINGLE_LINE_COMMENT : '//' [^]* -> skip;
```

Tuy nhiên, khi biên dịch một tập tin mã nguồn chứa nhiều dòng chú thích, chương trình đôi khi gặp lỗi hoặc hoạt động không như mong muốn. Hãy xác định các tình huống có thể gây ra lỗi với quy tắc trên và đề xuất một cách viết quy tắc tốt hơn để tránh lỗi trong ANTLR4.

Với quy tắc MULTI_LINE_COMMENT : '/*' .*? '*/' -> skip;, lexer sẽ hoạt động như sau:

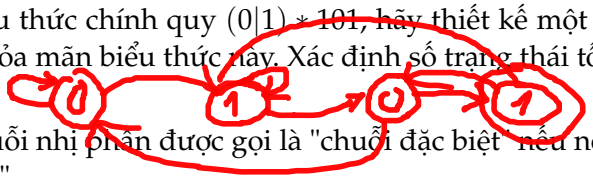
Khi gặp /* đầu tiên, nó bắt đầu khớp với quy tắc MULTI_LINE_COMMENT.

Nó sẽ khớp với mọi ký tự tiếp theo cho đến khi gặp */ đầu tiên, tức là */ ngay sau "chú thích đa dòng cấp 2 lồng bên trong".

Do đó, lexer sẽ chỉ nhận diện phần chú thích là /*\n Đây là chú thích đa dòng cấp 1\n /* Chú thích đa dòng cấp 2 lồng bên trong */.

Phần còn lại \n Và đây là phần còn lại của chú thích cấp 1\n */ và phần mã nguồn tiếp theo sẽ không được coi là chú thích nữa, và có thể gây ra lỗi cú pháp hoặc hành vi không mong muốn ở giai đoạn phân tích cú pháp (parsing) sau này.

Câu 05. Cho biểu thức chính quy $(0|1)*101$, hãy thiết kế một bộ tự động hữu hạn xác định (DFA) để nhận diện tập hợp chuỗi thỏa mãn biểu thức này. Xác định số trạng thái tối thiểu cần có trong DFA



Câu 06. Một chuỗi nhị phân được gọi là "chuỗi đặc biệt" nếu nó chứa đúng hai lần xuất hiện của dãy con "101" và không chứa "110".

- Hãy viết một biểu thức chính quy mô tả tập hợp chuỗi "đặc biệt" này.
- Xây dựng một bộ DFA tối thiểu để nhận diện các chuỗi "đặc biệt".

Tham lam (Greedy - mặc định): Các định lượng tham lam cố gắng khớp càng nhiều ký tự càng tốt. Chúng sẽ "ăn" càng nhiều ký tự trong chuỗi input càng tốt trong khi vẫn đảm bảo phần còn lại của mẫu regex có thể khớp thành công.

Không tham lam (Non-greedy/Reluctant/Lazy): Các định lượng không tham lam (thường được tạo ra bằng cách thêm dấu ? vào sau định lượng tham lam, ví dụ *?, +?, ??, {m,n}?) cố gắng khớp càng ít ký tự càng tốt. Chúng sẽ "ăn" ít ký tự nhất có thể trong khi vẫn đảm bảo phần còn lại của mẫu regex có thể khớp thành công.

Chương 3: PHÂN TÍCH NGỮ PHÁP

Câu 01. Cho các loại lỗi lập trình sau:

- Một biến được sử dụng mà chưa khai báo. Lỗi ngữ nghĩa
- Một hàm được gọi với sai số lượng tham số. Lỗi ngữ nghĩa
- Một phép toán chia một số cho 0. Lỗi thực thi
- Một vòng lặp vô hạn mà không có điều kiện dừng. Lỗi runtime
- Một cú pháp sai do quên dấu ; trong C++. Lỗi cú pháp

Hãy phân loại các lỗi trên thành lỗi cú pháp, lỗi ngữ nghĩa, hoặc lỗi runtime (thời gian chạy) và giải thích lý do cho từng loại lỗi.

Câu 02. Trong một ngôn ngữ lập trình, một số toán tử có thể được định nghĩa theo văn phạm sau:

$$E \to E + E \mid E * E \mid (E) \mid id$$

(a) Giải thích tại sao văn phạm này bị nhập nhằng.

(b) Hãy đề xuất cách chỉnh sửa văn phạm để đảm bảo độ ưu tiên toán tử (* có độ ưu tiên cao hơn +), nhưng vẫn giữ đúng ngữ nghĩa của ngôn ngữ.

Câu 03. Cho văn phạm phi ngữ cảnh sau: $S \to aSbS|\epsilon$

(a) Chứng minh rằng văn phạm này sinh ra tất cả các chuỗi có số lượng ký tự `a` bằng số lượng ký tự `b`, nhưng không nhất thiết phải ở dạng $a^n b^n$.

(b) Xây dựng **quy nạp toán học** để chứng minh rằng mọi chuỗi sinh ra từ văn phạm trên có thể được biểu diễn dưới dạng **chuỗi Dyck**, tức là một chuỗi dấu ngoặc hợp lệ trong đó `"a"` đóng vai trò dấu mở và `"b"` đóng vai trò dấu đóng.

(c) Chứng minh rằng văn phạm trên **không nhập nhằng** bằng cách chỉ ra rằng mọi chuỗi hợp lệ có **duy nhất một cây phân tích cú pháp**.

Câu 04. Xét hai văn phạm phi ngữ cảnh sau:

Văn phạm 1

$$S \to aSbc \mid abc$$

aabcbc

$S \Rightarrow a^{n+1}(bc)^{n+1}$.

Văn phạm 2

$$S \to abSc \mid abc$$

ababcc

$S \Rightarrow (ab)^{n+1}c^{n+1}$.

(a) Hai văn phạm này có sinh ra cùng một ngôn ngữ hay không? Giải thích bằng cách so sánh tập hợp chuỗi mà mỗi văn phạm có thể tạo ra. hai văn phạm không sinh ra cùng một ngôn ngữ.

(b) Nếu có sự khác biệt, hãy điều chỉnh một trong hai văn phạm để cả hai cùng sinh ra một ngôn ngữ duy nhất.

(a) Vai trò của ràng buộc động (Dynamic Binding) trong hỗ trợ tính đa hình
Định nghĩa: Ràng buộc động (hay còn gọi là "late binding") là cơ chế quyết định thời điểm gọi phương thức nào dựa trên kiểu thực (runtime type) của đối tượng, thay vì dựa trên kiểu khai báo (compile-time type).
Hỗ trợ đa hình:
Tính linh hoạt: Nhờ ràng buộc động, khi ta có một biến tham chiếu kiểu của lớp cơ sở (superclass) nhưng trỏ đến đối tượng của lớp dẫn xuất (subclass), việc gọi các phương thức sẽ được xác định tại thời điểm chạy dựa trên đối tượng thực sự. Điều này cho phép các đối tượng khác nhau (dù cùng kiểu khai báo) phản ứng khác nhau khi cùng nhận một lời gọi phương thức.
Ưu điểm: Cho phép viết mã tổng quát, dễ bảo trì và mở rộng, bởi các lớp dẫn xuất có thể ghi đè (override) các phương thức của lớp cơ sở để cung cấp hành vi cụ thể của riêng chúng.
Kết luận: Ràng buộc động là yếu tố cốt lõi để hiện thực đa hình trong lập trình hướng đối tượng vì nó cho phép liên kết giữa lời gọi phương thức và định nghĩa cụ thể của phương thức đó xảy ra tại thời điểm chạy.

Chương 4: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ LẬP TRÌNH HÀM

Câu 01. Trong lập trình hướng đối tượng, ràng buộc động (Dynamic Binding) và đa hình (Polymorphism) có mối liên hệ mật thiết với nhau.

(a) Hãy giải thích vai trò của ràng buộc động trong việc hỗ trợ tính đa hình.

(b) Nếu một ngôn ngữ lập trình không hỗ trợ ràng buộc động, thì cách duy nhất để hỗ trợ đa hình là gì? Đưa ra ví dụ minh họa bằng mã giả hoặc một ngôn ngữ lập trình cụ thể.

cách duy nhất để hỗ trợ đa hình là đa hình tĩnh (Static Polymorphism).
ví dụ: Function overloading hoặc template

Nạp chồng hàm (Function Overloading): Các hàm có cùng tên nhưng khác về tham số sẽ được lựa chọn khi biên dịch dựa trên kiểu và số lượng đối số.
Mẫu (Templates) hoặc Generic Programming: Cho phép viết các hàm hay lớp mà kiểu dữ liệu của đối số được xác định khi biên dịch.

Câu 02. Một số ngôn ngữ như C++ cho phép cả ràng buộc tĩnh và ràng buộc động, trong khi các ngôn ngữ như Python thực hiện ràng buộc động hoàn toàn.

Tính chất: Trong Python, mọi lời gọi phương thức đều được ràng buộc động theo mặc định. Nghĩa là, khi bạn gọi một phương thức trên một đối tượng, phiên bản của phương thức được quyết định tại thời điểm chạy (runtime) dựa trên kiểu thực của đối tượng.

(a) Hãy so sánh cách hoạt động của ràng buộc động trong Python với cách hoạt động của từ khóa virtual trong C++.

Nếu một phương thức trong lớp cơ sở không được khai báo bằng từ khóa virtual nhưng lại được lớp dẫn xuất ghi đè (override), thì:
Hiệu ứng: Khi gọi phương thức thông qua con trỏ hoặc tham chiếu kiểu lớp cơ sở, C++ sẽ thực hiện ràng buộc tĩnh (static binding) và gọi phiên bản của lớp cơ sở, thay vì gọi phiên bản của lớp dẫn xuất. Điều này làm mất đi tính đa hình mong muốn.
Tác động đến đa hình: Quá trình đa hình không được thực hiện động vì kết quả lời gọi phụ thuộc vào kiểu khai báo của biến chứ không phải kiểu thực của đối tượng.

(b) Điều gì sẽ xảy ra nếu một phương thức ảo (virtual method) trong C++ không được khai báo bằng từ khóa virtual? Nó có ảnh hưởng đến tính đa hình không? Giải thích bằng ví dụ.

Từ khóa virtual trong C++:
Tính chất: Trong C++, mặc định các phương thức được ràng buộc tĩnh (static binding), tức là lời gọi phương thức được quyết định tại thời điểm biên dịch dựa trên kiểu khai báo của biến. Khi một phương thức được khai báo với từ khóa virtual trong lớp cơ sở, thì phương thức đó được ràng buộc động, nghĩa là lời gọi phương thức sẽ được quyết định tại thời điểm chạy dựa trên kiểu thực của đối tượng.

Câu 03. C++ có các mức độ truy cập (`public`, `protected`, `private`), trong khi Python không có từ khóa kiểm soát truy cập chính thức mà chỉ dựa vào quy ước (`_protected`, `__private`).

(a) Tại sao Python không bắt buộc các mức độ truy cập như C++?

(b) Hãy viết hai đoạn mã trong C++ và Python để minh họa cách truy cập thuộc tính `private` từ bên ngoài lớp. Python có thực sự ngăn chặn được truy cập như C++ không?

```
class MyClass:
  def __init__(self, value):
    self.__value = value  # Thuộc tính được xem là private

  def get_value(self):
    return self.__value

obj = MyClass(42)

# Cố gắng truy cập trực tiếp vào __value sẽ gây lỗi:
try:
  print(obj.__value)
except AttributeError as e:
  print("Lỗi:", e)

# Tuy nhiên, ta vẫn có thể truy cập được thông qua name mangling:
print(obj._MyClass__value)  # In ra 42
```

Câu 04. Xét đoạn mã Python sau và :

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

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

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

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

D()
```

Trả lời các câu hỏi:

(a) Kết quả của chương trình trên là gì? DBCA

(b) Giải thích thứ tự gọi các phương thức __init__() của các lớp theo MRO.

(c) Nếu thay super().__init__() trong B và C bằng A.__init__(self), kết quả có thay đổi không? Nếu có, hãy giải thích. DBA

Câu 05.

Xét đoạn mã sau:

```
x = (1, 2, 3)
y = x
x += (4, 5)
```

y sẽ trỏ đến cùng một đối tượng tuple với x (1, 2, 3) - Khi thực hiện x += (4, 5), khi đó sẽ thực hiện lệnh cộng lên x tạo thành tuple (1, 2, 3, 4, 5) và gán cho x, nhưng y vẫn giữ nguyên tuple (1, 2, 3)

vì tuple là immutable (không thay đổi được), nên biểu thức x += (4, 5) tương đương với x = x + (4, 5). Điều này tạo ra một tuple mới (1,2,3,4,5) và gán cho biến x.

(a) Biến y có bị thay đổi sau khi thực hiện lệnh `x += (4, 5)` không?

(b) Nếu x là một danh sách thay vì tuple (`x = [1, 2, 3]`), kết quả có khác không? Giải thích.

(c) Dựa vào câu trên, hãy mô tả sự khác biệt giữa **thay đổi nội dung của một đối tượng mutable** và **gán lại một biến immutable**.

với danh sách (mutable), toán tử += (trong trường hợp này là phương thức list.__iadd__) sẽ thay đổi nội dung của danh sách tại chỗ (in-place) bằng cách thêm các phần tử từ danh sách [4, 5] vào cuối danh sách ban đầu.
Kết quả:
Danh sách được sửa thành [1, 2, 3, 4, 5].
Vì y cùng trỏ đến danh sách đó, nên y cũng phản ánh sự thay đổi này.
Như vậy, với danh sách, cả x và y đều có giá trị [1, 2, 3, 4, 5].
Sự khác biệt giữa thay đổi nội dung của một đối tượng mutable và gán lại một biến immutable
Đối với đối tượng immutable (ví dụ: tuple, int, str):
Khi bạn "gán lại" một biến (như x += (4,5)), một đối tượng mới được tạo ra và biến đó sẽ trỏ đến đối tượng mới.
Các biến khác trỏ đến đối tượng ban đầu sẽ không thay đổi.
Ví dụ: Ở (a), y vẫn giữ giá trị (1,2,3) trong khi x đã chuyển sang (1,2,3,4,5).
Đối với đối tượng mutable (ví dụ: list, dict, set):
Các phương thức hay toán tử (như +=) thường được cài đặt để thay đổi nội dung của đối tượng tại chỗ (in-place) mà không tạo đối tượng mới.
Vì nhiều biến có thể tham chiếu đến cùng một đối tượng, nên thay đổi nội dung sẽ được phản ánh trên tất cả các biến đó.
Ví dụ: Ở (b), cả x và y đều tham chiếu đến danh sách, nên khi nội dung của danh sách thay đổi, cả hai đều bị ảnh hưởng.

Câu 06. Cho đoạn mã sau:

```
a = (1, 2, [3, 4])
a[2].append(5)
print(a)
```

(a) Kết quả của chương trình trên là gì? Giải thích tại sao chúng ta có thể thay đổi nội dung của a, mặc dù a là một tuple.

(b) Làm thế nào để tạo một tuple hoàn toàn immutable, ngay cả khi nó chứa danh sách?

Điểm quan trọng: Tuple a không trực tiếp chứa các giá trị 1, 2, 3, 4. Thay vào đó, tuple a chứa tham chiếu đến các đối tượng này trong bộ nhớ. Trong đó, 1 và 2 là các đối tượng immutable (số nguyên), còn [3, 4] là một đối tượng mutable (danh sách).

a[2].append(5): Dòng này thực hiện hai việc:

a[2]: Truy cập vào phần tử thứ ba của tuple a. Như đã nói ở trên, phần tử thứ ba này là một danh sách [3, 4]. a[2] trả về tham chiếu đến danh sách này.
.append(5): Phương thức append() là một phương thức thay đổi danh sách tại chỗ (in-place modification). Nó thêm phần tử 5 vào danh sách mà a[2] tham chiếu tới.
Vì danh sách là mutable, chúng ta có thể thay đổi nội dung của nó thông qua tham chiếu a[2]. Tuy nhiên, bản thân tuple a không hề bị thay đổi. a vẫn giữ nguyên các tham chiếu ban đầu của nó: tham chiếu đến số 1, tham chiếu đến số 2, và tham chiếu đến cùng một danh sách (mà nội dung của danh sách này đã bị thay đổi).

Làm thế nào để tạo một tuple hoàn toàn immutable, ngay cả khi nó chứa danh sách?

Để tạo một tuple hoàn toàn immutable, ngay cả khi nó chứa các cấu trúc dữ liệu tương tự như danh sách, bạn cần đảm bảo rằng tất cả các phần tử bên trong tuple, và tất cả các phần tử bên trong các phần tử đó (nếu có), đều là immutable.

Trong trường hợp danh sách, cách đơn giản nhất để làm cho nó immutable là chuyển đổi danh sách thành một tuple. Vì tuple là immutable.

Câu 07. Currying và function composition là hai khái niệm quan trọng trong lập trình hàm.

(a) Hãy giải thích sự khác biệt giữa **currying** và **function composition** trong Python.

(b) Viết một đoạn mã Python sử dụng **currying function** để tạo một hàm `add_prefix(prefix)` thêm tiền tố vào một chuỗi, và sau đó sử dụng **function composition** để kết hợp `add_prefix("Mr.")` với một hàm `to_uppercase()` để tạo ra hàm `shout_with_prefix(name)`.

(c) Khi nào nên sử dụng **currying** thay vì **function composition**, và ngược lại?

Currying:

Mục tiêu: Currying là kỹ thuật chuyển đổi một hàm nhận nhiều đối số thành một chuỗi các hàm, mỗi hàm chỉ nhận một đối số duy nhất.
Cách thức hoạt động: Thay vì gọi hàm một lần với tất cả các đối số, bạn gọi hàm ban đầu với đối số đầu tiên. Kết quả trả về là một hàm mới, hàm này "ghi nhớ" đối số đầu tiên và sẵn sàng nhận đối số thứ hai. Quá trình này tiếp tục cho đến khi tất cả các đối số được cung cấp, cuối cùng hàm sẽ thực hiện tính toán và trả về kết quả cuối cùng.
Tính năng quan trọng: Currying cho phép partial application (ứng dụng một phần). Bạn có thể cung cấp một số đối số cho hàm và tạo ra một hàm mới "chuyên biệt" hơn, đã được cấu hình sẵn với các đối số đó. Hàm mới này sau đó có thể được sử dụng lại với các đối số còn lại.
Ví dụ: Xét hàm add(x, y) cộng hai số. Currying hàm này sẽ tạo ra một chuỗi các hàm như sau:
curried_add = curry(add) (giả sử curry là một hàm thực hiện currying)
add_5 = curried_add(5) (add_5 bây giờ là một hàm chỉ cần nhận một đối số y và trả về 5 + y)
result = add_5(3) (result sẽ là 8)

Function Composition (Hợp thành hàm):

Mục tiêu: Function composition là kỹ thuật kết hợp nhiều hàm lại với nhau để tạo thành một hàm phức tạp hơn.
Cách thức hoạt động: Bạn có một loạt các hàm, giả sử f, g, và h. Composition sẽ tạo ra một hàm mới bằng cách áp dụng các hàm này theo thứ tự: hợp_thành(f, g, h)(x) sẽ tương đương với f(g(h(x))). Nghĩa là, đầu tiên h được áp dụng cho x, kết quả sau đó được đưa vào g, và cuối cùng kết quả của g được đưa vào f.
Tính năng quan trọng: Function composition giúp xây dựng các quy trình xử lý dữ liệu tuần tự một cách rõ ràng và mạch lạc. Nó tăng cường tính module hóa và khả năng tái sử dụng code, vì bạn có thể kết hợp các hàm nhỏ, đơn giản để tạo ra các chức năng phức tạp hơn.
Ví dụ: Xét hàm to_uppercase(s) chuyển chuỗi thành chữ hoa và hàm remove_whitespace(s) loại bỏ khoảng trắng. Composition có thể tạo ra một hàm mới process_string(s) thực hiện cả hai việc:
process_string = compose(to_uppercase, remove_whitespace) (giả sử compose là hàm thực hiện composition)
result = process_string(" hello world ") (result sẽ là "HELLOWORLD")

```
from functools import partial

# Hàm gốc để thêm tiền tố
def add_prefix_func(prefix, text):
    return prefix + " " + text

# Currying để tạo hàm add_prefix nhận tiền tố trước
add_prefix = partial(add_prefix_func)

# Hàm chuyển chuỗi thành chữ hoa
def to_uppercase(text):
    return text.upper()

# Function composition (tự định nghĩa hàm compose đơn giản cho 2 hàm)
def compose(f, g):
    return lambda x: f(g(x))

# Kết hợp add_prefix("Mr.") và to_uppercase để tạo shout_with_prefix
# Lưu ý: Chúng ta cần partial(add_prefix_func, "Mr.") để cố định prefix "Mr." trước khi compose
shout_with_prefix = compose(to_uppercase, partial(add_prefix_func, "Mr."))

# Sử dụng các hàm đã tạo
add_mr = add_prefix("Mr.") # Tạo hàm add_mr chuyên biệt để thêm "Mr."
print(add_mr("John Smith")) # Output: Mr. John Smith

shout_name = shout_with_prefix("Jane Doe")
print(shout_name) # Output: MR. JANE DOE
```

dùng "partial" để thực hiện currying (cho phép nhận 1 đối số trước khi nhận cái còn lại, thay vì phải nhận cả 2 cùng lúc)

Sử dụng currying function khi:
Tạo các phiên bản chuyên biệt của 1 hàm (add5(x), add7(x)...
Muốn trì hoãn việc cung cấp một đối số
Tăng tính linh hoạt + khả năng cấu hình của hàm
Các callback/xử lí sự kiện

Sử dụng function compisition khi:
Muốn xây dựng một trình tự xử lí dữ liệu (f(g(x))...
Tái sử dụng các hàm nhỏ để xây dựng các hàm phức tạp hơn
Cải thiện cấu trúc source code
Kết hợp các hàm đã tồn tại để tạo ra hành vi mới

Tổng quan:

Currying tập trung vào việc điều chỉnh và cấu hình một hàm duy nhất thông qua partial application.
Function Composition tập trung vào việc kết hợp và sắp xếp nhiều hàm để tạo ra quy trình xử lý logic tuần tự.

Câu 08. Python cho phép sử dụng **nhiều decorators** trên một hàm.

(a) Viết một đoạn mã trong đó một hàm được trang bị hai decorators:

- `uppercase`: Chuyển đổi output thành chữ in hoa.
- `times(n)`: Lặp lại việc gọi hàm `n` lần.

(b) Khi thực thi hàm, kết quả sẽ bị ảnh hưởng như thế nào bởi thứ tự khai báo decorators?

```
def uppercase(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def times(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            output = ""
            for _ in range(n):
                output += func(*args, **kwargs)
            return output
        return wrapper
    return decorator

@uppercase
@times(3)
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))
```

Thứ tự decorators quan trọng: Decorator được liệt kê gần định nghĩa hàm hơn (ở đây là @times(3)) sẽ được áp dụng đầu tiên. Sau đó, decorator ở trên (@uppercase) sẽ được áp dụng tiếp theo cho kết quả của decorator bên dưới. Như vậy, @times(3) được áp dụng trước, sau đó @uppercase được áp dụng cho kết quả của @times(3).

=> HELLO, ALICE!HELLO, ALICE!HELLO, ALICE!

Giả sử:

```
@times(3)
@uppercase
def greet(name):
    return f"Hello, {name}!"

print("\nTrường hợp 2:")
print(greet("Alice"))
```

=> Vẫn trả về "HELLO, ALICE!HELLO, ALICE!HELLO, ALICE!", nhưng là upper trước rồi mới nahan lên 3 lần

Trong cả hai trường hợp, kết quả cuối cùng dường như giống nhau: "HELLO, ALICE!HELLO, ALICE!HELLO, ALICE!". Tuy nhiên, đây chỉ là sự trùng hợp trong ví dụ cụ thể này.

Chương 5: SINH CÂY CÚ PHÁP TRỪU TƯỢNG

Câu 01. Trong quá trình biên dịch, **Parse Tree** và **AST** đóng vai trò quan trọng trong phân tích cú pháp và tạo mã trung gian.

(a) Hãy giải thích sự khác biệt chi tiết giữa **Parse Tree** và **AST**, tập trung vào cấu trúc, mục đích và mức độ trừu tượng .

(b) Viết một ví dụ với biểu thức toán học `(3 + 4) * 5`, vẽ **Parse Tree** và **AST** tương ứng.

(c) Tại sao trình biên dịch thường sử dụng **AST** thay vì **Parse Tree** trong các giai đoạn sau của quá trình biên dịch?

Bảng so sánh tóm tắt:

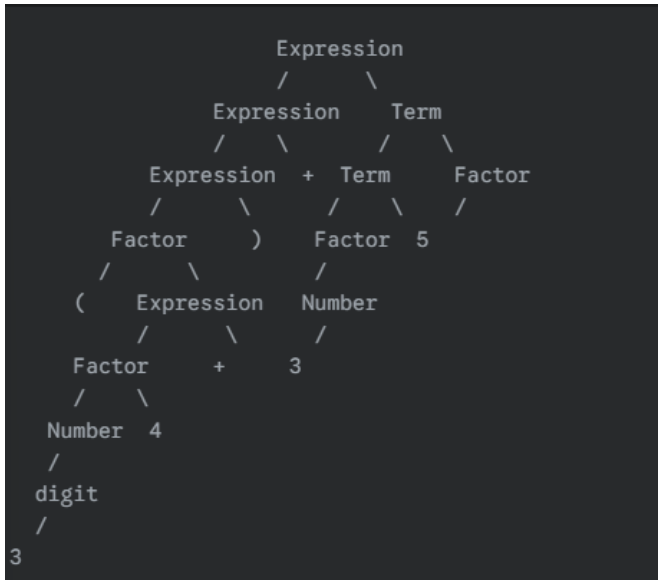
Đặc điểm	Parse Tree (Cây Phân Tích Cú Pháp)	AST (Cây Cú Pháp Trừu Tượng)
Tên gọi khác	Concrete Syntax Tree (CST)	Abstract Syntax Tree (AST)
Cấu trúc	Chi tiết, đầy đủ, phản ánh ngữ pháp	Tối giản, trừu tượng, tập trung ý nghĩa
Node đại diện cho	Ký hiệu ngữ pháp (terminal & non-terminal)	Cấu trúc ngôn ngữ lập trình (biểu thức, lệnh)
Mục đích	Minh họa phân tích cú pháp, xác minh cú pháp	Biểu diễn trung gian, phục vụ phân tích ngữ nghĩa & sinh mã
Mức độ trừu tượng	Thấp (low-level), cụ thể (concrete)	Cao (high-level), trừu tượng (abstract)
Kích thước	Lớn, phức tạp	Nhỏ gọn, đơn giản
Quan hệ với ngữ pháp	Phản ánh trực tiếp quy tắc ngữ pháp	Không phản ánh trực tiếp quy tắc ngữ pháp

b)

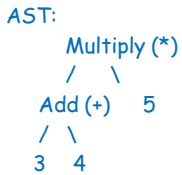
Giả sử ta có:

Expression -> Term | Expression + Term
Term -> Factor | Term * Factor
Factor -> Number | (Expression)
Number -> digit | digit Number
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Parse tree:



AST:



Tại sao trình biên dịch thường sử dụng AST thay vì Parse Tree trong các giai đoạn sau của quá trình biên dịch?

Trình biên dịch thường sử dụng AST thay vì Parse Tree trong các giai đoạn sau của quá trình biên dịch (như phân tích ngữ nghĩa, sinh mã trung gian, tối ưu hóa mã, sinh mã đích) vì những lý do chính sau:

Đơn giản và Trừu tượng:

AST đơn giản và trừu tượng hơn Parse Tree. Nó loại bỏ các chi tiết cú pháp không cần thiết, giúp các giai đoạn xử lý tiếp theo dễ dàng hơn.
Parse Tree chứa quá nhiều thông tin cú pháp cụ thể (ví dụ: các ký hiệu ngữ pháp trung gian, dấu câu, từ khóa) mà không đóng góp nhiều vào ý nghĩa ngữ nghĩa của chương trình. Việc xử lý một cấu trúc phức tạp như Parse Tree sẽ kém hiệu quả hơn.
Phân tích Ngữ nghĩa Hiệu quả hơn:

AST được thiết kế để tập trung vào ý nghĩa ngữ nghĩa của chương trình. Cấu trúc của nó trực tiếp phản ánh các cấu trúc ngôn ngữ lập trình quan trọng (biểu thức, câu lệnh, khai báo).
Các bước phân tích ngữ nghĩa (như kiểm tra kiểu dữ liệu, phân tích phạm vi biến, kiểm tra tính hợp lệ của ngữ nghĩa) trở nên đơn giản và trực tiếp hơn khi làm việc với AST. Ví dụ, việc kiểm tra kiểu của một biểu thức cộng trong AST dễ dàng hơn nhiều so với việc dò tìm trong Parse Tree phức tạp.
Sinh Mã và Tối ưu hóa Mã Dễ dàng hơn:

AST cung cấp một biểu diễn logic chương trình rõ ràng, giúp cho việc sinh mã trung gian và mã máy dễ dàng hơn.
Các thuật toán tối ưu hóa mã (ví dụ: loại bỏ mã chết, tối ưu hóa biểu thức) có thể được áp dụng hiệu quả hơn trên AST vì nó đã loại bỏ các chi tiết cú pháp không liên quan và làm nổi bật cấu trúc chương trình.
Việc chuyển đổi từ AST sang mã trung gian hoặc mã máy trở nên trực tiếp hơn vì AST đã có cấu trúc gần với cách mà máy tính thực thi chương trình.
Tính Độc lập Tương đối với Ngữ pháp:

AST ít phụ thuộc vào ngữ pháp cụ thể của ngôn ngữ hơn so với Parse Tree. AST tập trung vào cấu trúc logic và ý nghĩa, trong khi Parse Tree bám sát vào các quy tắc ngữ pháp.
Điều này có nghĩa là trình biên dịch có thể sử dụng cùng một cấu trúc AST cho các ngôn ngữ có ngữ pháp khác nhau nhưng có ý nghĩa ngữ nghĩa tương tự. Hoặc khi ngữ pháp của ngôn ngữ có sự thay đổi, AST có thể ít bị ảnh hưởng hơn so với Parse Tree.
Hiệu suất và Bộ nhớ:

Do AST đơn giản và nhỏ gọn hơn Parse Tree, việc lưu trữ và xử lý AST yêu cầu ít bộ nhớ và tính toán hơn. Điều này đặc biệt quan trọng đối với các trình biên dịch lớn và các chương trình phức tạp.
Tóm lại: AST là một bản tóm tắt thông minh và hiệu quả của cấu trúc chương trình, được thiết kế đặc biệt để phục vụ cho các giai đoạn biên dịch sau giai đoạn phân tích cú pháp. Bằng cách loại bỏ các chi tiết cú pháp rườm rà và tập trung vào ý nghĩa ngữ nghĩa, AST giúp quá trình biên dịch trở nên hiệu quả hơn, dễ bảo trì và dễ mở rộng hơn. Parse Tree vẫn đóng vai trò quan trọng trong giai đoạn đầu của quá trình biên dịch (phân tích cú pháp), nhưng AST mới là cấu trúc dữ liệu chính được sử dụng trong các giai đoạn tiếp theo.