TRƯỜNG ĐẠI HỌC
SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
HCMC University of Technology and Education
HCMUTE

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN CUỐI KỲ

***

XÂY DỰNG COMPILER ĐƠN GIẢN

GVHD: ĐINH CÔNG ĐOAN
MÃ HP: SYPR432780

SINH VIÊN THỰC HIỆN

STT	HỌ VÀ TÊN	MSSV
1	Nguyễn Minh Tâm	22162039

Thành phố Hồ Chí Minh, Tháng 4 Năm 2025

MỤC LỤC

Phần 1: Phần mở đầu

1.1. Tóm tắt đề tài

Trong lĩnh vực công nghệ phần mềm, trình biên dịch đóng vai trò thiết yếu trong việc chuyển đổi mã nguồn từ ngôn ngữ bậc cao sang mã máy – bước trung gian để chương trình có thể thực thi được trên phần cứng. Nhằm mục tiêu tiếp cận và thấu hiểu toàn diện quá trình này, nhóm nghiên cứu đã thực hiện đề tài "Xây dựng trình biên dịch đơn giản bằng ngôn ngữ Python để biên dịch ngôn ngữ C", tập trung vào việc hiện thực hóa các thành phần nền tảng nhất của một trình biên dịch, bao gồm: bộ phân tích từ vựng, bộ phân tích cú pháp, bộ phân tích ngữ nghĩa, và bộ sinh mã.

Thông qua việc sử dụng ngôn ngữ Python – một ngôn ngữ dễ đọc, dễ phát triển và được hỗ trợ bởi các thư viện mạnh mẽ như ply (Python Lex-Yacc) – đề tài không chỉ giúp làm rõ các khái niệm lý thuyết trong môn Lập trình hệ thống mà còn cung cấp mô hình thực tiễn để sinh viên trực tiếp trải nghiệm việc xây dựng một công cụ biên dịch có tính khả thi. Python cho phép đơn giản hóa việc triển khai các thành phần như lexer và parser, đồng thời phù hợp cho việc prototyping nhanh. Kết quả của đề tài là một chương trình có khả năng đọc mã nguồn C đơn giản, phân tích cú pháp, và sinh ra mã hợp ngữ tương ứng, tạo tiền đề cho việc mở rộng về sau như hỗ trợ thêm cấu trúc ngôn ngữ, tối ưu hóa mã, hoặc tích hợp với các công cụ phân tích hiện đại.

1.2. Đặt vấn đề nghiên cứu

1.2.1. Tổng quan nghiên cứu liên quan

Lĩnh vực xây dựng trình biên dịch đã được nghiên cứu chuyên sâu trong nhiều thập kỷ qua, với những đóng góp nền tảng từ các công trình học thuật kinh điển. Nổi bật nhất phải kể đến tác phẩm "Compilers: Principles, Techniques, and Tools" của nhóm tác giả Aho, Lam, Sethi và Ullman - được giới chuyên môn xem như "kinh thánh" trong lĩnh vực này. Bên cạnh đó, các dự án thực tiễn như TinyCC hay LCC cùng hàng trăm dự án mã nguồn mở trên nền tảng GitHub đã cung cấp những minh chứng cụ thể về quá trình triển khai từng thành phần của trình biên dịch trong môi trường thực tế.

1.2.2. Tính cấp thiết của đề tài

Trong bối cảnh công nghệ thông tin phát triển như vũ bão hiện nay, việc nắm vững nguyên lý hoạt động của trình biên dịch không còn là yêu cầu thuần túy lý thuyết đối với sinh viên công nghệ thông tin. Kiến thức này đã trở thành nền tảng thiết yếu cho nhiều lĩnh vực ứng dụng quan trọng như phát triển ngôn ngữ lập trình mới, kỹ thuật dịch ngược phần mềm, hay các giải pháp bảo mật hệ thống ở mức độ sâu. Chính vì vậy, việc tự thiết kế và xây dựng một trình biên dịch đơn giản không chỉ là cơ hội quý giá để áp dụng kiến thức lý thuyết vào thực tiễn, mà còn giúp phát triển tư duy hệ thống - kỹ năng không thể thiếu của một kỹ sư phần mềm chuyên nghiệp.

1.2.3. Cơ sở tài liệu nghiên cứu

Để đảm bảo tính khoa học và độ tin cậy của đề tài, nhóm nghiên cứu đã tham khảo hệ thống tài liệu đa dạng bao gồm:

(1) Các giáo trình chuyên sâu về nguyên lý trình biên dịch,

(2) Tài liệu hướng dẫn kỹ thuật lập trình hệ thống bằng ngôn ngữ Python,

(3) Mã nguồn mở của các trình biên dịch đơn giản như TINY hay MINICC, cùng

(4) Các khóa học trực tuyến từ các nền tảng MOOC uy tín như Coursera và edX.

Sự kết hợp giữa tài liệu học thuật và nguồn thực tiễn này tạo thành nền móng vững chắc cho quá trình nghiên cứu.

1.2.4. Lý do lựa chọn đề tài

Việc lựa chọn đề tài "Xây dựng trình biên dịch đơn giản bằng ngôn ngữ Python để biên dịch ngôn ngữ C" được định hướng dựa trên ba yếu tố then chốt, kết hợp cả nhu cầu học thuật lẫn năng lực thực hành chuyên môn.

Trước hết, đây là một đề tài tiêu biểu cho sự giao thoa giữa nền tảng lý thuyết và ứng dụng thực tiễn. Quá trình xây dựng trình biên dịch đòi hỏi người học phải vận dụng kiến thức từ nhiều lĩnh vực – như ngôn ngữ hình thức, lập trình hệ thống, tổ chức máy tính – để giải quyết một bài toán phức hợp, có tính hệ thống cao. Chính điều này tạo ra môi trường học tập đa chiều, nơi lý thuyết không còn mang tính trừu tượng mà được chuyển hóa thành các dòng mã có thể kiểm chứng được hiệu quả.

Thứ hai, việc lựa chọn ngôn ngữ lập trình Python không chỉ xuất phát từ tính phổ biến, mà còn từ đặc tính kỹ thuật của nó. Python có cú pháp rõ ràng, dễ đọc, và được hỗ trợ bởi các thư viện mạnh mẽ như ply, giúp đơn giản hóa việc xây dựng các thành phần phức tạp như lexer và parser. Điều này cho phép người học tập trung vào việc hiểu nguyên lý hoạt động của trình biên dịch thay vì xử lý các chi tiết kỹ thuật cấp thấp, đồng thời phù hợp cho việc phát triển nhanh và kiểm thử.

Cuối cùng, đề tài mở ra cơ hội quan trọng để rèn luyện kỹ năng nghiên cứu chuyên sâu, cụ thể là khả năng đọc hiểu tài liệu học thuật, tài liệu kỹ thuật tiếng Anh, phân tích mã nguồn mở, và làm việc với các công cụ hỗ trợ chuyên nghiệp như ply hoặc NASM. Đây đều là những kỹ năng thiết yếu không chỉ cho nghiên cứu khoa học mà còn trong môi trường làm việc kỹ thuật cao sau này.

Tổng hòa cả ba yếu tố trên cho thấy rằng đề tài không chỉ mang lại giá trị học thuật thuần túy, mà còn góp phần bồi dưỡng tư duy hệ thống và năng lực phát triển phần mềm, phù hợp với định hướng đào tạo kỹ sư công nghệ thông tin chuyên sâu.

1.2.5. Mục tiêu nghiên cứu

Về mặt học thuật, đề tài hướng đến mục tiêu xây dựng thành công một trình biên dịch tối giản nhưng đầy đủ các thành phần cốt lõi: bộ phân tích từ vựng (lexical analyzer), bộ phân tích cú pháp (syntax parser), cơ chế xây dựng cây cú pháp trừu tượng (AST), và hệ thống sinh mã trung gian. Về mặt kỹ năng, quá trình thực hiện sẽ giúp nâng cao năng lực nghiên cứu tài liệu chuyên ngành, phát triển tư duy hệ thống, và kỹ năng làm việc với các công cụ lập trình hệ thống chuyên sâu.

1.2.6. Phạm vi và giới hạn nghiên cứu

Để đảm bảo tính khả thi, đề tài tập trung vào việc xây dựng trình biên dịch cho một ngôn ngữ lập trình đơn giản có cú pháp tương đồng với ngôn ngữ C, bao gồm các cấu trúc cơ bản như khai báo biến, câu lệnh điều kiện, vòng lặp và các phép toán số học. Các vấn đề nâng cao như tối ưu hóa mã, xử lý lỗi phức tạp, hay hỗ trợ các kiểu dữ liệu phức tạp sẽ không nằm trong phạm vi nghiên cứu lần này. Đầu ra của trình biên dịch được giới hạn ở mã trung gian hoặc mã giả có thể kiểm chứng thông qua máy ảo hoặc trình thông dịch đơn giản.

1.2.7. Phương pháp nghiên cứu

Về mặt lý thuyết, phương pháp phân tích - tổng hợp được sử dụng để hệ thống hóa các nguyên lý cốt lõi. Trên phương diện thực tiễn, quá trình phát triển sử dụng ngôn ngữ Python cùng thư viện hỗ trợ chuyên nghiệp như ply (Python Lex-Yacc) để xây dựng bộ phân tích từ vựng và cú pháp. Mỗi thành phần của hệ thống sẽ được kiểm thử độc lập theo phương pháp unit testing trước khi tích hợp toàn hệ thống, đảm bảo tính chính xác ở mọi giai đoạn phát triển.

1.2.8. Cấu trúc nội dung

Nội dung báo cáo được tổ chức thành bốn phần chính. Phần đầu tiên trình bày tổng quan về kiến thức nền tảng và nguyên lý hoạt động của trình biên dịch. Phần thứ hai đi sâu vào phân tích cấu trúc hệ thống và thiết kế các module chức năng. Phần thứ ba tập trung vào quá trình triển khai cụ thể từ lexical analysis đến code generation. Cuối cùng, phần đánh giá kết quả sẽ cung cấp các chỉ số đo lường cụ thể về hiệu quả hoạt động của hệ thống cùng những bài học kinh nghiệm rút ra từ quá trình thực hiện.

Phần 2: Phần nội dung

Chương 1: Giới thiệu

1.1. Tổng quan về compiler và vai trò trong Lập trình Hệ thống

Trình biên dịch (compiler) là một phần mềm hệ thống có nhiệm vụ chuyển đổi mã nguồn được viết bằng ngôn ngữ lập trình bậc cao thành mã máy hoặc mã trung gian mà hệ thống có thể thực thi một cách trực tiếp hoặc gián tiếp. Đây là thành phần cốt lõi trong chuỗi công cụ phát triển phần mềm, giúp kết nối giữa tư duy lập trình cấp cao của con người và quá trình xử lý ở mức phần cứng.

Trong bối cảnh môn học *Lập trình Hệ thống*, trình biên dịch có vai trò đặc biệt quan trọng vì nó minh họa rõ nét sự giao thoa giữa phần mềm và phần cứng, giữa trừu tượng và hiện thực. Việc nghiên cứu và xây dựng một trình biên dịch không chỉ giúp sinh viên hiểu rõ quy trình xử lý ngôn ngữ lập trình mà còn tạo điều kiện tiếp cận các khái niệm sâu hơn về tối ưu hóa mã, quản lý bộ nhớ, và hoạt động của hệ điều hành ở mức thấp.

Việc tự tay triển khai một trình biên dịch, dù chỉ là ở mức độ đơn giản, sẽ giúp người học hình dung được toàn bộ quá trình biến đổi của chương trình – từ các dòng mã nguồn do con người viết ra cho đến các chuỗi lệnh máy tính thực thi. Đây cũng là nền tảng quan trọng để hiểu được cách thức hoạt động của các hệ thống nhúng, các phần mềm hiệu năng cao, cũng như cơ chế tối ưu hóa tại tầng thấp của máy tính.

1.2. Mục tiêu của đề tài

Mục tiêu chính của đề tài là thiết kế và triển khai một trình biên dịch cơ bản cho ngôn ngữ C, tập trung vào việc hỗ trợ một tập con đơn giản của ngôn ngữ này, đồng thời thực hiện toàn bộ quá trình biên dịch bằng ngôn ngữ lập trình Python. Trình biên dịch sẽ chuyển đổi mã nguồn C thành tệp thực thi thông qua các giai đoạn cốt lõi: phân tích từ vựng (lexical analysis) để tách mã nguồn thành các đơn vị từ vựng (token), phân tích cú pháp (syntax analysis) để xây dựng cấu trúc cú pháp của chương trình, tạo cây cú pháp trừu tượng (Abstract Syntax Tree - AST) nhằm biểu diễn cấu trúc logic, sinh mã trung gian dưới dạng mã hợp ngữ x86-64, và cuối cùng sử dụng bộ hợp dịch (assembler) cùng bộ liên kết (linker) để tạo tệp thực thi chạy được trên hệ thống.

Về mặt học thuật, trình biên dịch được xây dựng mang tính minh họa, giúp làm sáng tỏ các thành phần và cơ chế hoạt động của một compiler thực thụ, từ đó cung cấp một công cụ trực quan để nghiên cứu và giảng dạy về lập trình hệ thống. Việc lựa chọn Python làm ngôn ngữ phát triển tận dụng cú pháp linh hoạt, dễ đọc, cùng các thư viện mạnh mẽ như re (cho xử lý regex trong lexer) và ply (cho phân tích cú pháp), giúp đẩy nhanh quá trình phát triển và cho phép tập trung vào các khía cạnh lý thuyết thay vì chi tiết kỹ thuật cấp thấp.

Để tăng tính thực tiễn, trình biên dịch được tích hợp với một giao diện đồ họa (GUI) sử dụng Tkinter, cho phép người dùng nhập mã nguồn, lưu/mở tệp, xem mã hợp ngữ được sinh ra, và thực thi chương trình một cách trực quan. GUI không chỉ nâng cao trải nghiệm người dùng mà còn minh họa rõ ràng mối liên hệ giữa mã nguồn và đầu ra thực thi, hỗ trợ mục tiêu giáo dục của đề tài.

Do giới hạn về thời gian và phạm vi nghiên cứu, trình biên dịch chỉ hỗ trợ một tập con tối giản của ngôn ngữ C, bao gồm các kiểu dữ liệu cơ bản (int, string), khai báo biến, câu lệnh điều kiện (if-else), vòng lặp (while), phép toán số học đơn giản, và các hàm nhập/xuất cơ bản (printf, scanf). Các chương trình mẫu nhỏ, chẳng hạn như tính toán số học hoặc xử lý điều kiện, sẽ được sử dụng để kiểm chứng khả năng của trình biên dịch. Mặc dù không thể sánh ngang với các trình biên dịch thương mại như GCC về tính năng hay hiệu suất, sản phẩm này vẫn mang lại giá trị lớn trong việc làm sáng tỏ nguyên lý hoạt động bên trong của một trình biên dịch, đồng thời cung cấp nền tảng để mở rộng các tính năng nâng cao trong tương lai, chẳng hạn như hỗ trợ hàm con, biểu thức phức tạp hoặc tối ưu hóa mã.

Bên cạnh đó, đề tài hướng đến việc phát triển kỹ năng thực hành của nhóm, bao gồm khả năng làm việc với các công cụ lập trình hệ thống, nghiên cứu tài liệu kỹ thuật, và tích hợp các thành phần phần mềm thành một hệ thống hoàn chỉnh. Kết quả cuối cùng không chỉ là một trình biên dịch hoạt động mà còn là một bài học thực tiễn về cách các khái niệm lý thuyết được áp dụng để giải quyết các vấn đề kỹ thuật phức tạp, từ đó góp phần nâng cao năng lực nghiên cứu và phát triển phần mềm của người thực hiện.

Chương 2: Phân tích yêu cầu

2.1. Yêu cầu của compiler cơ bản

Để đạt được mục tiêu đề ra, trình biên dịch C cơ bản cần đáp ứng các yêu cầu sau:

- **Ngôn ngữ nguồn**: Hỗ trợ một tập con đơn giản của ngôn ngữ C. Cụ thể, mã nguồn đầu vào luôn có hàm main duy nhất với kiểu trả về int. Bên trong main, cho phép khai báo biến kiểu int hoặc string và thực hiện một số thao tác đơn giản trên các biến này. Hỗ trợ các câu lệnh gán, biểu thức số học (+, -, *, /), câu lệnh điều kiện if-else, câu lệnh vào/ra cơ bản printf và scanf, và câu lệnh return trả về số nguyên. Không yêu cầu hỗ trợ con trỏ (trừ trường hợp sử dụng toán tử & trước biến trong scanf), mảng, cấu trúc, hay hàm do người dùng định nghĩa (chỉ có hàm main duy nhất).
- **Mã đích**: Sinh ra mã hợp ngữ x86-64 (NASM syntax) cho hệ điều hành Linux (định dạng đối tượng ELF 64-bit). Từ mã hợp ngữ này có thể hợp dịch thành mã máy (object file) và liên kết với thư viện chuẩn để tạo thành file thực thi 64-bit chạy trên Linux.
- **Tích hợp quá trình dịch**: Trình biên dịch cần tích hợp hoặc hỗ trợ việc gọi trình hợp dịch NASM và trình liên kết (GCC) để tự động hóa quá trình biên dịch hoàn chỉnh. Người dùng có thể cung cấp tệp mã nguồn C và nhận được tệp thực thi sau khi trình biên dịch chạy qua tất cả các bước.
- **Thông báo lỗi**: Cần phát hiện một số lỗi cú pháp cơ bản trong mã nguồn và thông báo cho người dùng (ví dụ: thiếu dấu ;, sai từ khoá, v.v.). Mặc dù không yêu cầu quá cao như compiler thực thụ, chương trình nên có cơ chế báo lỗi khi gặp cấu trúc không hỗ trợ hoặc sai định dạng đầu vào.

Tóm lại, yêu cầu đặt ra là xây dựng một trình biên dịch có khả năng dịch đúng các chương trình C đơn giản (theo phạm vi hỗ trợ) thành mã máy chạy được, qua đó kiểm chứng tính đúng đắn của quá trình dịch.

2.2. Các thành phần cần triển khai

Để đáp ứng các yêu cầu trên, hệ thống trình biên dịch sẽ bao gồm các thành phần chính sau

- **Bộ phân tích từ vựng (Lexer)**: Thành phần này chịu trách nhiệm đọc mã nguồn dạng văn bản và tách thành các đơn vị từ vựng cơ bản gọi là *token*. Mỗi token đại diện cho một thực thể ý nghĩa như từ khoá (int, if, else, ...), định danh, toán tử, hằng số, dấu câu, khoảng trắng, v.v. Bộ phân tích từ vựng cần phân loại đúng loại token và bỏ qua những thành phần không cần thiết (như khoảng trắng, xuống dòng, comment nếu có). Kết quả đầu ra của giai đoạn này là một danh sách các token làm đầu vào cho bộ phân tích cú pháp.
- **Bộ phân tích cú pháp (Parser)**: Sử dụng danh sách token từ lexer, parser sẽ kiểm tra cấu trúc câu lệnh của chương trình dựa trên văn phạm C đơn giản. Parser xây dựng một cây cú pháp trừu tượng (AST - Abstract Syntax Tree) để biểu diễn cấu trúc lô-gic của chương trình. Mỗi nút (node) trong AST thể hiện một thành phần cú pháp (ví dụ: một phép gán, một biểu thức điều kiện, một câu lệnh if, v.v.). Giai đoạn này đảm bảo rằng chuỗi token đầu vào sắp xếp theo thứ tự hợp lệ (ví dụ đúng cú pháp của ngôn ngữ) và lưu trữ cấu trúc đó trong bộ nhớ để xử lý tiếp.
- **Bộ sinh mã hợp ngữ (Code Generator)**: Dựa trên AST được xây dựng, thành phần này duyệt cây và sinh ra mã hợp ngữ tương ứng. Mã hợp ngữ đầu ra tuân theo các quy tắc của kiến trúc đích (ở đây là tập lệnh x86-64) và tuân theo các quy ước lời gọi (calling convention) cần thiết để tương tác với thư viện C (chẳng hạn sử dụng thanh ghi phù hợp cho hàm printf, scanf, và đảm bảo thiết lập stack frame cho hàm main). Kết quả của bước này là một tệp mã nguồn hợp ngữ (*.asm*) có thể dịch bằng assembler.
- **Trình hợp dịch (Assembler)**: Sử dụng công cụ NASM để chuyển đổi mã hợp ngữ (.asm) thành mã máy dạng nhị phân (object file .o). Quá trình này đảm bảo các lệnh hợp ngữ, nhãn, hằng, v.v. được mã hóa thành mã máy thực thi, đồng thời tạo các bảng thông tin cần thiết cho việc liên kết (như bảng kí hiệu cho các biến và hàm extern).
- **Trình liên kết (Linker)**: Sử dụng công cụ liên kết (ở đây dùng GCC như một trình bao gồm liên kết) để kết hợp object file ở trên với các thư viện cần thiết (ví dụ thư viện

chuẩn libc cung cấp hàm printf, scanf) và tạo ra file thực thi cuối cùng. Linker sẽ xử lý việc nối các đoạn mã (code section, data section), xử lý liên kết động tới các hàm thư viện, và tạo điểm bắt đầu chương trình (entry point) để có thể chạy được.

- **Môi trường thực thi**: Để compiler hoạt động trọn vẹn, hệ thống cần có môi trường phù hợp: cài đặt Python 3 để chạy mã nguồn trình biên dịch, cài đặt NASM (Netwide Assembler) để hợp dịch mã, và GCC (hoặc linker tương đương) để liên kết. Hệ thống thí điểm ở đây là Linux 64-bit, do đó các công cụ trên cần có sẵn trên Linux.

Các thành phần trên sẽ được tổ chức thành các mô-đun Python riêng biệt tương ứng: module lexer/parser/codegen (tích hợp trong một file Python chính), module assembler (gọi NASM) và module linker (gọi GCC). Sự phối hợp nhịp nhàng giữa các thành phần đảm bảo quá trình biên dịch diễn ra tự động, cho phép chuyển từ mã nguồn C sang chương trình thực thi chỉ bằng một lệnh chạy trình biên dịch.

Chương 3: Những kiến thức liên quan

3.1. Khái niệm cơ bản về trình biên dịch và các giai đoạn

Một trình biên dịch hoàn chỉnh thường được chia thành hai giai đoạn lớn: *giai đoạn phân tích* (analysis/front-end) và *giai đoạn tổng hợp* (synthesis/back-end). Giai đoạn phân tích nhằm đọc và hiểu chương trình nguồn, còn giai đoạn tổng hợp sẽ tạo ra chương trình đích (mã máy) tương ứng. Cụ thể, các bước chính của quá trình biên dịch bao gồm:

- **Phân tích từ vựng (Lexical Analysis)**: Là bước đầu tiên của giai đoạn front-end. Ở bước này, mã nguồn (chuỗi ký tự) được quét tuần tự và nhóm thành các đơn vị từ vựng gọi là *token*. Quá trình này loại bỏ các khoảng trắng, xuống dòng và chú thích, đồng thời phân loại các chuỗi ký tự thành token có ý nghĩa (như từ khóa int, danh định (identifier) main, toán tử =, số 100, v.v.). Kết quả là một dãy các token sẽ được chuyển cho bộ phân tích cú pháp. *Lexical analysis* giúp đơn giản hóa công việc của phân tích cú pháp bằng cách trừu tượng hóa đầu vào thành các đơn vị cơ bản.
- **Phân tích cú pháp (Syntax Analysis)**: Bước này nhận luồng token từ lexer và kiểm tra xem chuỗi token đó có tuân theo văn phạm của ngôn ngữ hay không. Một *bộ phân tích cú pháp* sẽ xây dựng cây cú pháp trừu tượng (AST) để biểu diễn cấu trúc lô-gic của chương trình. AST là một cây trong đó mỗi nút đại diện cho một thành phần ngữ nghĩa trong chương trình (ví dụ nút gán giá trị, nút biểu thức, nút câu lệnh điều kiện, ...). Khác với *cây phân tích cụ thể (parse tree)* chứa mọi chi tiết cú pháp (dấu ngoặc, dấu chấm phẩy,...), AST lược bỏ chi tiết dư thừa và chỉ giữ lại cấu trúc ngữ nghĩa cốt lõi. Giai đoạn này cũng bao gồm một mức độ phân tích ngữ nghĩa nhất định, như kiểm tra kiểu dữ liệu, kiểm tra khai báo (biến phải được khai báo trước khi sử dụng), v.v. (Trong phạm vi trình biên dịch đơn giản này, việc kiểm tra ngữ nghĩa sẽ rất hạn chế, hầu hết logic kiểm tra được tích hợp ngay trong quá trình parse). Nếu gặp lỗi cú pháp (ví dụ thiếu dấu }, token không hợp lệ ở một vị trí nào đó), trình biên dịch sẽ báo lỗi và dừng quá trình.

- **Tạo mã trung gian (Intermediate Code Generation)**: Trong các compiler phức tạp, sau khi có AST, trình biên dịch thường chuyển AST thành mã trung gian độc lập với máy (ví dụ mã bytecode, hoặc mã ba địa chỉ). Mã trung gian là biểu diễn mức thấp hơn AST nhưng chưa phải mã máy cụ thể, giúp cho việc tối ưu hóa và chuyển đổi sang nhiều kiến trúc khác nhau dễ dàng hơn (Chương trình dịch là gì? Phân loại chương trình dịch?). Tuy nhiên, trong trình biên dịch C đơn giản này chúng ta bỏ qua bước mã trung gian, thay vào đó sinh thẳng mã hợp ngữ. Lý do: phạm vi nhỏ không đòi hỏi tối ưu phức tạp, và việc sinh trực tiếp mã hợp ngữ giúp đơn giản hóa quá trình.
- **Tối ưu hóa mã (Code Optimization)**: Đây là bước tùy chọn nhưng quan trọng trong các compiler hiện đại. Trình biên dịch sẽ cố gắng cải tiến mã trung gian hoặc mã máy để chạy hiệu quả hơn (ví dụ: loại bỏ mã dư thừa, giảm số lệnh, tận dụng thanh ghi, ...). Tối ưu hóa có thể diễn ra trên mã trung gian (như hợp nhất các biểu thức hằng, giản lược biểu thức, loại bỏ đoạn mã chết) hoặc trên mã máy (sắp xếp lại lệnh để tận dụng pipeline CPU, v.v.). Trong dự án này, bước tối ưu hóa sẽ không được triển khai sâu do mục đích chính là tạo ra trình biên dịch hoạt động đúng chức năng. Mã sinh ra sẽ ở mức đúng chức năng, chưa tối ưu hiệu suất.
- **Sinh mã đích (Code Generation)**: Giai đoạn tổng hợp cuối cùng, chuyển mã trung gian (hoặc AST) thành mã máy đích. Với trình biên dịch cơ bản này, giai đoạn này chính là việc sinh mã hợp ngữ x86-64. Trình sinh mã sẽ duyệt AST, với mỗi nút tương ứng sinh ra một hoặc một loạt các lệnh hợp ngữ tương đương. Ví dụ, một phép gán a = b + 1; có thể được sinh thành các lệnh máy: nạp giá trị của b vào thanh ghi, cộng 1, rồi lưu kết quả vào ô nhớ của a. Quá trình sinh mã cần tuân thủ chặt chẽ các quy tắc của kiến trúc CPU (tập lệnh, thanh ghi) cũng như quy ước gọi hàm của hệ điều hành (đặc biệt khi gọi các hàm thư viện như printf, scanf).
- **Hợp dịch (Assembling) và Liên kết (Linking)**: Mặc dù không luôn được coi là một phần của "trình biên dịch" nội tại, nhưng đây là các bước cần thiết để từ mã hợp ngữ có thể thành chương trình hoàn chỉnh. *Assembler* chuyển mã hợp ngữ (dạng text) thành mã máy dạng nhị phân (object file) – công việc này thẳng tiến và tương đối đơn giản (mỗi lệnh assembly trở thành một mã nhị phân tương ứng). *Linker* sau đó kết hợp các

object file (bao gồm cả object file sinh ra từ mã của chúng ta và các object file thư viện) thành file thực thi. Linker sẽ gán địa chỉ thực cho các nhãn, hàm, biến, và đảm bảo tất cả lời gọi hàm bên ngoài đều được nối kết đến định nghĩa của chúng (ví dụ lời gọi printf được liên kết đến mã thực sự của printf trong thư viện chuẩn C). Kết quả cuối cùng là một file thực thi ELF 64-bit có thể chạy trên Linux.

Các khái niệm trên tạo nền tảng cho việc thiết kế trình biên dịch. Hiểu rõ từng giai đoạn giúp việc cài đặt (implementation) trở nên mạch lạc: ta có thể phát triển, kiểm thử từng thành phần (lexer, parser, codegen, ...) độc lập trước khi tích hợp chúng lại với nhau.

3.2. Các đặc tả về ngôn ngữ C áp dụng trong compiler

Trình biên dịch mẫu này hướng đến một *tập con rất giới hạn của ngôn ngữ C*. Việc giới hạn này nhằm đơn giản hóa quá trình xử lý, giúp tập trung vào những ý chính trong xây dựng compiler mà không tập trung quá nhiều vào chi tiết ngôn ngữ.

Dưới đây là các đặc điểm ngôn ngữ C được hỗ trợ và những giới hạn tương ứng:

- **Cấu trúc chương trình**: Bắt buộc có hàm int main() {...} là điểm bắt đầu chương trình. Không hỗ trợ nhiều hàm hay hàm có tham số. Toàn bộ mã nguồn hợp lệ phải nằm bên trong thân hàm main. Không hỗ trợ khai báo hàm tùy ý khác ngoài main.
- **Kiểu dữ liệu và biến**: Hỗ trợ khai báo biến kiểu int (số nguyên 32-bit) và char (ký tự 1 byte). Tuy nhiên, do hạn chế của trình code generator, cả int và char trong hiện thực này đều được lưu trữ trên 8 byte (64-bit) trong *section .data* của chương trình (sử dụng chỉ thị dq – định nghĩa một *quad word* 8 byte cho mỗi biến). Điều này nghĩa là trình biên dịch không phân biệt kích thước thật sự giữa int và char, coi chúng đều như số nguyên 64-bit để đơn giản việc xử lý trên thanh ghi 64-bit. Biến phải được khai báo đầu khối { } của hàm main (sau dòng int main() {), và có thể khởi tạo giá trị ban đầu. Nếu không khởi tạo thì mặc định sẽ được gán giá trị 0. Ví dụ:
    - int a = 5;
    - char c;
    - int x = 0;

Các biến này sẽ được cấp phát tĩnh trong vùng dữ liệu .data của chương trình dịch, chứ không phải trên stack (do chưa triển khai cấp phát biến cục bộ trên stack frame).

- **Biểu thức và phép toán**: Hỗ trợ các phép toán số học cơ bản: +, -, *, / giữa các toán hạng kiểu nguyên (hoặc char, vốn cũng coi như số nguyên nhỏ). Mỗi biểu thức trong phiên bản này chỉ được phép có một toán tử nhị phân (do parser đơn giản chỉ xử lý một phép một lúc). Tức là các biểu thức phức tạp phải được chia nhỏ thành nhiều phép gán tuần tự.

Ví dụ, biểu thức a + b - 3 không được phân tích trực tiếp; người dùng phải viết thành hai bước:

  - int t = a + b;
  - int result = t - 3;

  Parser hiện tại chưa xử lý được thứ tự ưu tiên hay kết hợp của toán tử; nó đọc một phép tính dạng X op Y duy nhất tại bên phải dấu =. Ngoài ra, có hỗ trợ toán tử so sánh trong mệnh đề if (xem bên dưới) nhưng cũng giới hạn tương tự (một phép so sánh duy nhất). Các toán tử được lexer nhận diện gồm: +, -, *, /, ==, !=, <, >, <=, >=, ++, --, và toán tử & (dùng trong scanf). Tuy nhiên, trình biên dịch **chỉ thực sự xử lý/sinh mã** cho một số trường hợp: + - * / trong biểu thức gán, và một phần của > trong biểu thức điều kiện (các toán tử khác có thể được nhận diện token nhưng chưa được sinh mã đúng). Đây là một điểm hạn chế: ví dụ, sử dụng == trong điều kiện if sẽ vẫn được parser chấp nhận nhưng code sinh ra sẽ không đúng (vì code generator của if mặc định thực hiện kiểm tra dạng >). Do vậy, người lập trình sử dụng trình biên dịch này cần tuân theo một số quy ước ngầm phù hợp với cách sinh mã hiện tại (ví dụ: chỉ dùng > trong điều kiện if nếu muốn có kết quả đúng).

- **Câu lệnh gán (assignment)**: Cho phép gán giá trị cho biến với cú pháp var = expr; trong đó expr là một biểu thức số học đơn giản hoặc hằng hoặc biến. Bên trái dấu = phải là một biến đã khai báo. Bên phải có thể là:

- Một hằng số nguyên (ví dụ x = 5;).
- Một biến khác (ví dụ y = x; được parser diễn giải như y = x + 0 do parse logic luôn tìm một toán tử, trường hợp này toán tử ngầm định là cộng 0).
- Một phép tính giữa hai toán hạng (ví dụ a = b + 1;, c = a - b;). Như đã nói, parser không hỗ trợ nhiều hơn một toán tử.

Mỗi câu lệnh gán phải kết thúc bằng ;. Parser triển khai sẽ tách câu lệnh gán ra thành các thành phần: tên biến đích, toán hạng trái, toán tử, toán hạng phải. Những thành phần này sau đó được lưu trong một nút AST kiểu Assign để code generator tạo mã.

- **Cấu trúc điều kiện if-else**: Hỗ trợ câu lệnh điều kiện dạng:

```
if (cond) {
   // các câu lệnh gán
} else {
   // các câu lệnh gán
}
```

Trong đó cond là một điều kiện so sánh đơn giản dạng x <op> y với <op> có thể là >, <, >=, <=, ==, != (theo lexer). Tuy nhiên, như đã lưu ý, code generator hiện tại chỉ xử lý đúng trường hợp điều kiện > (lớn hơn) giữa hai toán hạng. Cú pháp yêu cầu có cặp ngoặc tròn bao quanh điều kiện và cặp ngoặc nhọn bao quanh khối then/else. Phần else { ... } là tùy chọn (có thể không có else). Bên trong mỗi nhánh if hoặc else, chỉ cho phép các câu lệnh gán đơn giản. Thiết kế parser hiện tại không cho phép lồng các câu lệnh if khác hoặc gọi printf, scanf bên trong if. Nếu người dùng cố viết if lồng nhau hoặc đặt printf bên trong if, parser sẽ không hiểu đúng (vì trong ngữ cảnh đó nó chỉ chờ đợi các lệnh gán). Giới hạn này giúp parser đơn giản hơn rất nhiều (vì không phải đệ quy xử lý nhiều loại câu lệnh bên trong block). Dù vậy, nó cũng giảm tính biểu đạt của ngôn ngữ nguồn.

- **Hàm thư viện hỗ trợ I/O**: Cho phép sử dụng printf và scanf để in ra màn hình và đọc vào từ bàn phím. Cú pháp hỗ trợ:
    - printf("chuoi dinh dang", x); với một chuỗi định dạng và tối đa một tham số (có thể là biến, hằng số số nguyên hoặc hằng ký tự). Ví dụ: printf("x = %d", x); để in giá trị của x. Nếu chuỗi định dạng không yêu cầu tham số (không chứa %d), có thể gọi printf chỉ với chuỗi (ví dụ printf("Hello");). Parser sẽ kiểm tra và chấp nhận trường hợp có hoặc không có đối số thứ hai. Hiện tại, trình biên dịch giả định chỉ dùng %d trong chuỗi định dạng để in số nguyên hoặc ký tự (in ký tự cũng dùng %d sẽ in mã ASCII, chưa hỗ trợ %c).
    - scanf("%d", &x); để đọc một số nguyên vào biến x. Cú pháp yêu cầu chuỗi định dạng (hiện tại giả định luôn là "%d") và đối số là &<biến>. Parser sẽ kiểm tra bắt buộc ký tự & ngay trước tên biến, nếu thiếu sẽ báo lỗi. Cũng tương tự, chưa hỗ trợ đọc ký tự bằng %c hay các kiểu khác.

Những hàm này thực chất không được triển khai trực tiếp trong trình biên dịch, mà được gọi vào thư viện chuẩn. Code generator sẽ sinh lệnh gọi tới printf và scanf (được khai báo extern). Do đó, khi liên kết cần đảm bảo liên kết với thư viện C (điều mà GCC sẽ tự làm). Hỗ trợ printf/scanf giúp chương trình dịch có thể tương tác cơ bản với người dùng, làm cho các ví dụ trực quan hơn (thay vì chỉ tính toán khô khan).

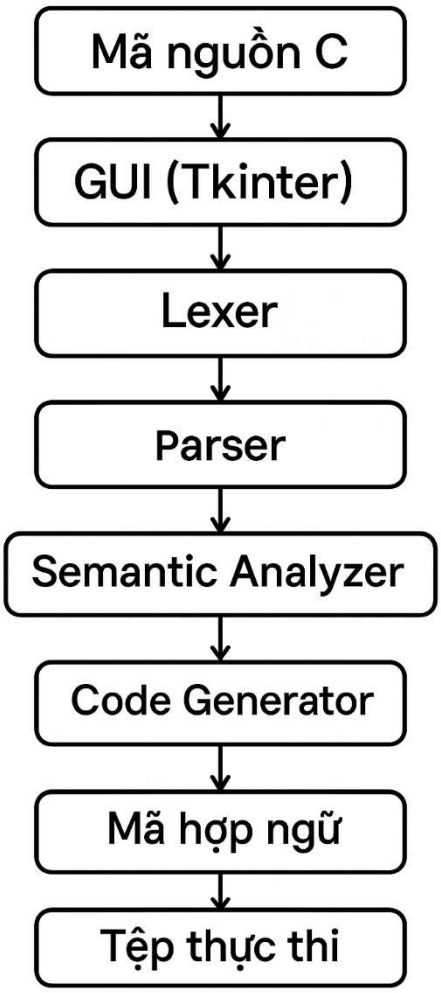
- **Câu lệnh return**: Hỗ trợ return <số nguyên>; ở cuối hàm main để trả về mã kết thúc chương trình. Nếu parser gặp return sẽ lấy token tiếp theo nếu là số thì dùng làm giá trị trả về, nếu không có hoặc không phải số thì mặc định trả về 0. Thực tế, trong C chuẩn, return 0; trong main có thể được bỏ qua (mặc định hệ thống coi giá trị trả về = 0 nếu không có return cuối hàm main). Nhưng ở đây, để đơn giản, ví dụ mẫu thường sẽ có return 0;. Trình biên dịch không cho phép return ở vị trí tùy ý (chưa kiểm soát điều này), nhưng thường người dùng chỉ đặt ở cuối.
- **Các yếu tố không hỗ trợ khác**: Không hỗ trợ vòng lặp (for, while), không hỗ trợ biểu thức phức tạp hoặc gọi hàm tùy ý, không quản lý phạm vi biến cục bộ phức tạp (biến chỉ có trong main và coi như "toàn cục" suốt chương trình). Ngoài ra, không phân biệt

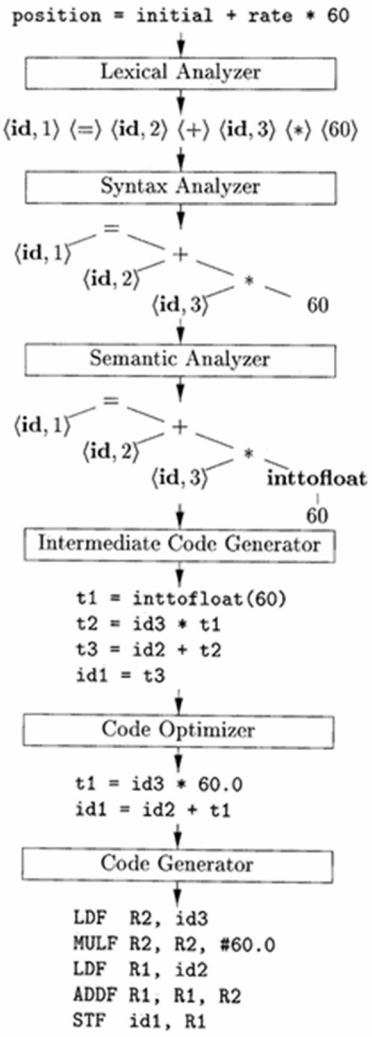
được trường hợp biến chưa khởi tạo sử dụng (trình biên dịch không cảnh báo nếu dùng biến chưa gán giá trị, nó vẫn cấp phát trong .data và nếu không khởi tạo sẽ cho 0).

Nhìn chung, ngôn ngữ mà trình biên dịch này hỗ trợ tương đương với một ngôn ngữ giả C rất đơn giản, gần với ngôn ngữ máy. Mặc dù hạn chế, tập con này đủ để biểu diễn một số thuật toán nhỏ và đặc biệt là minh họa hành vi của trình biên dịch.

Chương 4: Ứng dụng

4.1. Sơ đồ khối





Nguyên tắc làm việc của hệ thống

Lexical Analyzer: Phân tích từ vựng, hay còn gọi là giai đoạn quét, là bước đầu tiên trong quá trình hoạt động của một trình biên dịch. Tại giai đoạn này, bộ phân tích từ vựng (lexer) xử lý luồng ký tự của chương trình nguồn, tổ chức chúng thành các chuỗi có ý nghĩa được gọi là lexeme (từ tố). Mỗi lexeme được chuyển đổi thành một token, có định dạng (tên_token, giá_trị_thuộc_tính), và được chuyển tiếp sang giai đoạn phân tích cú pháp.

- Tên_token: Là một biểu tượng trừu tượng, đại diện cho loại lexeme (ví dụ: từ khóa, toán tử, biến), được sử dụng trong quá trình phân tích cú pháp.
- Giá_trị_thuộc_tính: Là một tham chiếu đến một mục trong bảng ký hiệu, chứa thông tin chi tiết về token, chẳng hạn như giá trị hoặc kiểu dữ liệu.

Syntax analyzer: Phân tích cú pháp, hay parsing, là giai đoạn thứ hai của quá trình biên dịch, trong đó bộ phân tích cú pháp (parser) sử dụng các thành phần đầu tiên của các token từ giai đoạn phân tích từ vựng để tạo ra một biểu diễn trung gian dạng cây, được gọi là cây cú pháp trừu tượng (Abstract Syntax Tree - AST).

Semantic analyzer: Bộ phân tích ngữ nghĩa sử dụng cây cú pháp và thông tin trong bảng ký hiệu để kiểm tra tính nhất quán ngữ nghĩa của chương trình nguồn với định nghĩa của ngôn ngữ lập trình. Nó cũng thu thập thông tin về kiểu dữ liệu và lưu trữ thông tin này trong cây cú pháp hoặc bảng ký hiệu, để sử dụng trong giai đoạn tạo mã trung gian tiếp theo.

Intermediate Code Generator:

- Trong quá trình dịch một chương trình nguồn thành mã đích, trình biên dịch có thể tạo ra một hoặc nhiều biểu diễn trung gian, có thể ở nhiều dạng khác nhau. Cây cú pháp là một dạng biểu diễn trung gian phổ biến, thường được sử dụng trong giai đoạn phân tích cú pháp và phân tích ngữ nghĩa.
- Sau khi hoàn thành phân tích cú pháp và ngữ nghĩa của chương trình nguồn, nhiều trình biên dịch tạo ra một biểu diễn trung gian cấp thấp hoặc giống mã máy, được xem như một chương trình cho một máy trừu tượng. Biểu diễn trung gian này cần có hai đặc tính quan trọng: dễ tạo ra và dễ dịch sang mã máy đích.

Code Optimizer: Giai đoạn tối ưu hóa mã không phụ thuộc vào máy (machine-independent code optimization) tập trung vào việc cải thiện mã trung gian để tạo ra mã đích hiệu quả hơn. Mục tiêu chính là tăng tốc độ thực thi của chương trình đích, nhưng cũng có thể bao gồm các mục tiêu khác như giảm kích thước mã hoặc giảm mức tiêu thụ năng lượng.

Code Generator: Giai đoạn tạo mã (code generation) là bước cuối cùng trong quá trình biên dịch, trong đó bộ tạo mã chuyển đổi biểu diễn trung gian của chương trình nguồn thành mã trong ngôn ngữ đích, chẳng hạn như mã máy. Khi ngôn ngữ đích là mã máy, bộ tạo mã cần thực hiện các nhiệm vụ sau:

- Phân bổ tài nguyên: Lựa chọn các thanh ghi hoặc vị trí bộ nhớ phù hợp để lưu trữ các biến được sử dụng trong chương trình.
- Dịch lệnh: Chuyển đổi các lệnh trung gian (ví dụ: mã ba địa chỉ) thành chuỗi các lệnh máy thực hiện cùng chức năng.

Việc quan trọng là phân bổ thanh ghi sao cho hiệu quả, giúp tối ưu hóa hiệu suất của chương trình đích bằng cách giảm thiểu việc truy cập bộ nhớ và tăng tốc độ thực thi.

4.2. Các bước xây dựng

4.2.1. Xác định văn phạm và chức năng cần hỗ trợ:

Dựa trên mục tiêu và phân tích yêu cầu, xác định cụ thể cú pháp được chấp nhận của ngôn ngữ nguồn (C đơn giản). Từ đó, viết ra các quy tắc văn phạm cho các thành phần như cấu trúc if-else, câu lệnh gán, lời gọi printf, v.v. Đồng thời, lên kế hoạch cấu trúc dữ liệu (AST) để biểu diễn các thực thể ngữ pháp này trong code Python.

4.2.2. Xây dựng bộ phân tích từ vựng (Lexer):

Chọn cách thức nhận diện token. Ở đây quyết định sử dụng **biểu thức chính quy (Regular Expressions)** vì Python có sẵn thư viện re mạnh, phù hợp để nhận dạng mẫu từ vựng. Viết các biểu thức regex cho từng loại token (từ khóa, số, định danh, toán tử, ký tự, chuỗi, ký hiệu) và tổ chức hàm lexer để quét mã nguồn. Thử nghiệm hàm lexer độc lập trên một số chuỗi đơn giản để đảm bảo phân tách token đúng mong đợi.

4.2.3. Xây dựng bộ phân tích cú pháp (Parser):

Lựa chọn phương pháp phân tích cú pháp. Với văn phạm đơn giản, phương pháp đệ quy xuống (recursive descent) là phù hợp (cũng có thể gọi là parser thủ công). Triển khai các hàm con tương ứng với từng quy tắc ngữ pháp (ví dụ: parse_if, parse_assign, parse_printf, ...), sử dụng một biến toàn cục trỏ vị trí hiện tại trong danh sách token. Parser sẽ duyệt qua list token, đảm bảo tuần tự phù hợp với cú pháp và tạo các đối tượng AST tương ứng. Viết các lớp AST Node trong Python để chứa thông tin của các thành phần (như lớp Assign chứa tên biến và biểu thức gán, lớp If chứa điều kiện và danh sách câu lệnh then/else, v.v.). Kiểm thử parser trên những chuỗi token giả lập (hoặc kết hợp với lexer trên đoạn code mẫu) để xem AST thu được có đúng cấu trúc mong muốn.

4.2.4. Phát triển bộ sinh mã (Code generator):

Sau khi có AST, viết hàm duyệt qua AST (có thể là duyệt theo pre-order) để sinh mã hợp ngữ. Xác định chiến lược sinh mã cho từng loại nút AST:

- Với khai báo biến: sinh ra dòng trong section .data để cấp phát biến.
- Với nút gán: sinh ra chuỗi lệnh assembly để thực hiện phép gán (tùy theo toán tử là + - * /).
- Với nút if: sinh mã cho phần so sánh và nhảy có điều kiện, cũng như mã cho các phần then, else.
- Với nút gọi printf, scanf: sinh mã thiết lập tham số và gọi hàm tương ứng.
- Với nút return: sinh mã trả về (đưa giá trị vào thanh ghi eax và nhảy ra khỏi hàm). Tinh chỉnh mã sinh cho đúng cú pháp NASM và đúng quy ước gọi hàm hệ thống (ví dụ thứ tự thanh ghi cho tham số). In thử mã hợp ngữ ra tệp và kiểm tra bằng mắt xem hợp lý chưa. Có thể sử dụng NASM để hợp dịch thử và dùng objdump để kiểm tra mã máy (nếu cần).

4.2.5. Tích hợp GCC:

Tích hợp lệnh biên dịch gcc -no-pie vào chương trình để đầu ra là một mã máy. Đảm bảo script kiểm tra sự hiện diện của công cụ, gọi đúng tham số: gcc -no-pie -o program.s output.o. Thử nghiệm trên mã hợp ngữ sinh ra từ bước 4 để chắc chắn quá trình assembler/linker chạy suôn sẻ, tạo được file thực thi.

4.2.6. Kiểm thử end-to-end:

Chạy toàn bộ pipeline trên các chương trình C mẫu, từ lexer -> parser -> codegen -> assembler -> linker -> chạy chương trình. Đối chiếu kết quả chương trình chạy với kỳ vọng. Nếu có lỗi, quay lại các bước trên để sửa (ví dụ: sai cú pháp assembly, thiếu lệnh trong codegen, hoặc parser chấp nhận sai cấu trúc...). Thực hiện nhiều ví dụ từ đơn giản đến phức tạp trong phạm vi cho phép để đánh giá độ tin cậy của compiler.

4.2.7. Hoàn thiện và viết báo cáo:

Tổng kết lại kiến trúc, các quyết định thiết kế, hạn chế còn tồn tại và hướng mở rộng. Ghi lại mã nguồn cuối cùng và tài liệu hóa cho người dùng (cách sử dụng trình biên dịch, cú pháp được hỗ trợ, ...).

Những bước trên đảm bảo cách tiếp cận tuần tự, xây dựng dần dần một compiler từ đơn giản đến hoàn chỉnh. Trong quá trình thực hiện, chúng tôi tập trung trước vào tính đúng đắn (sinh mã chạy đúng), chưa ưu tiên tối ưu mã. Mỗi thành phần được kiểm thử độc lập trước khi tích hợp, giúp cô lập và sửa lỗi dễ dàng hơn.

4.3. Chi tiết triển khai các thành phần chính

Phần này đi sâu vào mã nguồn Python đã viết cho trình biên dịch, phân tích cách hiện thực từng chức năng: từ lexical analysis, parsing & AST, đến code generation. Trích xuất và giải thích những đoạn mã tiêu biểu để làm rõ cách hoạt động.

4.3.1. Bộ phân tích từ vựng (Lexer)

Đầu tiên, xem cách xác định mẫu token trong mã Python. Ta định nghĩa một danh sách các cặp *(loại_token, biểu_thức_regex)* gọi là TOKEN. Mỗi mục mô tả một loại token cần nhận diện và biểu thức chính quy tương ứng để bắt chuỗi ký tự thuộc loại đó. Dưới đây là danh sách token được định nghĩa trong c_asm.py

```
import re,sys

TOKENS = [
    # Kiểu dữ liệu cơ bản C: int hoặc string
    ('TYPE',           r'\b(int|string)\b'),
    # Các từ khóa điều khiển IO
    ('PRINTF',         r'\bprintf\b'),
    ('SCANF',          r'\bscanf\b'),
    # Các từ khóa điều khiển luồng
    ('IF',             r'\bif\b'),
    ('ELSE',           r'\belse\b'),
    ('WHILE',          r'\bwhile\b'),
    ('RETURN',         r'\breturn\b'),
    # Ký tự đặc biệt
    ('AMPERSAND',      r'&'),
    # Tên biến, hàm: bắt đầu bằng chữ hoặc gạch dưới, tiếp theo có thể là chữ/số/gạch dưới
    ('IDENTIFIER',     r'[a-zA-Z_][a-zA-Z0-9_]*'),
    # Số nguyên (liên tiếp các chữ số)
    ('NUMBER',         r'\d+'),
    # Xâu ký tự: mở bằng " và cho phép escape nội bộ
    ('STRING_LITERAL', r'"([^"\\]|\\.)*"'),
    # Các toán tử so sánh
    ('COMPARE_OP',     r'>=|<=|==|!=|>|<'),
    # Toán tử gán
    ('ASSIGN',         r'='),
    # Toán tử số học: + - * /
    ('ARITH_OP',       r'\+|-|\*|/'),
    # Dấu ngoặc, dấu ngoặc nhọn, chấm phẩy, phẩy
    ('LPAREN',         r'\('),
    ('RPAREN',         r'\)'),
    ('LBRACE',         r'\{'),
    ('RBRACE',         r'\}'),
    ('SEMI',           r';'),
    ('COMMA',          r','),
]
```

Các mẫu regex trong danh sách TOKENS được định nghĩa như sau:

- **TYPE**: Mẫu \b(int|string)\b nhận diện các từ khóa kiểu dữ liệu int và string. Ranh giới từ (\b) đảm bảo chỉ khớp các từ độc lập, tránh nhầm lẫn với các chuỗi như integer.
- **PRINTF, SCANF, IF, ELSE, WHILE, RETURN**: Các mẫu \bprintf\b, \bscanf\b, v.v., nhận diện các từ khóa điều khiển luồng và nhập/xuất. Tương tự, ranh giới từ đảm bảo khớp chính xác.

- **AMPERSAND**: Mẫu & nhận diện toán tử lấy địa chỉ, quan trọng trong các lệnh scanf (ví dụ: scanf("%d", &x);).
- **IDENTIFIER**: Mẫu [a-zA-Z_][a-zA-Z0-9_]* nhận diện định danh (tên biến, tên hàm) bắt đầu bằng chữ cái hoặc gạch dưới, theo sau là chữ cái, số, hoặc gạch dưới. Mẫu này được liệt kê sau các từ khóa để đảm bảo từ khóa được ưu tiên.
- **NUMBER**: Mẫu \d+ nhận diện các số nguyên không âm (ví dụ: 123). Lưu ý rằng số âm được xử lý bởi parser thông qua toán tử -.
- **STRING_LITERAL**: Mẫu "([^"\\]|\\.)*" nhận diện chuỗi ký tự trong dấu nháy kép, hỗ trợ ký tự escape (như \n, \"). Mẫu này đủ linh hoạt để xử lý các chuỗi như "%d" hoặc "hello\n".
- **COMPARE_OP**: Mẫu >=|<=|==|!=|>|< nhận diện các toán tử so sánh, với thứ tự ưu tiên cho các toán tử hai ký tự (>=, <=, v.v.) để tránh nhầm lẫn với toán tử một ký tự (>).
- **ASSIGN**: Mẫu = nhận diện toán tử gán.
- **ARITH_OP**: Mẫu \+|-|*|/ nhận diện các toán tử số học, sử dụng ký tự escape (\+, \-) để tránh nhầm lẫn với ký hiệu regex.
- **LPAREN, RPAREN, LBRACE, RBRACE, SEMI, COMMA**: Các mẫu \(, \) , \{ , \} , ; , , nhận diện các ký hiệu cú pháp quan trọng trong C.

Hàm lex(code) hoạt động như sau:

1. Loại bỏ khoảng trắng ở đầu chuỗi (code.lstrip()).
2. Với mỗi mẫu trong TOKENS, thử khớp với đầu chuỗi bằng re.match().
3. Nếu khớp, thêm token (tok_name, value) vào danh sách và cắt chuỗi nguồn tương ứng.
4. Nếu không khớp mẫu nào, ném ngoại lệ SyntaxError.

```python
def lex(code):
    tokens = []
    while code:
        code = code.lstrip()
        if not code:
            break
        for tok_name, tok_re in TOKENS:
            match = re.match(tok_re, code)
            if match:
                value = match.group(0)
                tokens.append((tok_name, value))
                code = code[len(value):]
                break
        else:
            raise SyntaxError(f"Unexpected character: {code[0]}")
    return tokens
```

Ví dụ: giả sử input là dòng:

int a = 10;

Lexer sẽ tạo các token:

[('KEYWORD','int'), ('IDENT','a'), ('OP','='), ('NUMBER','10'), ('SYMBOL',';')]

Mỗi cặp tuple gồm loại và nội dung chuỗi. Việc phân loại giúp parser dễ dàng nhận biết ngữ cảnh (ví dụ gặp '=' nhưng parser sẽ thấy loại OP và có thể kiểm tra giá trị cụ thể nếu cần).

Lexer không có cơ chế báo lỗi riêng nếu gặp chuỗi không thuộc về bất kỳ token nào trong TOK_SPEC. Trường hợp xấu nhất, những ký tự "lạ" sẽ không khớp mẫu nào và bị bỏ qua bởi finditer (do pattern WS hứng các ký tự trắng, các ký tự chữ cái, số đều đã có nhóm, chỉ các ký tự đặc biệt không liệt kê mới bị rơi). Tuy nhiên, do ngôn ngữ C hầu như mọi ký hiệu hợp lệ ta đã liệt kê nên xác suất có ký tự ngoài ý muốn (trừ lỗi chính tả của lập trình viên) là thấp. Nói chung, lexer đơn giản này đủ dùng cho tập con C mục tiêu.

4.3.2. Bộ phân tích cú pháp và cấu trúc AST

Parser (bộ phân tích cú pháp) chịu trách nhiệm kiểm tra xem danh sách token từ lexer có tuân theo ngữ pháp của tập con C hay không, đồng thời xây dựng một cây cú pháp trừu tượng (Abstract Syntax Tree - AST) để biểu diễn cấu trúc logic của chương trình. AST là một biểu diễn trung gian, giúp đơn giản hóa các bước phân tích ngữ nghĩa và sinh mã sau này.

Tính chính xác cú pháp: Parser trong compiler.py sử dụng kỹ thuật phân tích cú pháp đệ quy giảm dần (recursive descent parsing), được triển khai thông qua lớp Parser. Phương pháp này phù hợp với tập con C đơn giản vì nó dễ hiểu, dễ triển khai, và cho phép kiểm soát chi tiết quá trình phân tích.

Xử lý lỗi chi tiết: Parser cung cấp các thông báo lỗi cụ thể (như Expected SEMI, got ...) khi gặp token không mong đợi, giúp người dùng xác định vị trí lỗi cú pháp và sửa chữa nhanh chóng.

Tính linh hoạt: Cấu trúc của lớp Parser cho phép dễ dàng mở rộng để hỗ trợ các cấu trúc mới (như vòng lặp for hoặc hàm con) bằng cách thêm các phương thức phân tích mới mà không cần thay đổi toàn bộ mã.

Triển khai: Lớp Parser trong compiler.py được thiết kế để xử lý các cấu trúc của tập con C thông qua các phương thức phân tích như parse_statements, parse_declaration, parse_if_statement, v.v. Các phương thức chính bao gồm:

```
def parse_statements(self):
    statements = []
    while self.peek() and self.peek()[0] != 'RBRACE':
        next_tok = self.peek()[0]
        if next_tok == 'TYPE':
            statements.append(self.parse_declaration())  # Handle declarations
        elif next_tok == 'IDENTIFIER':
            statements.append(self.parse_assignment())  # Handle assignments
        elif next_tok == 'PRINTF':
            statements.append(self.parse_printf_statement())  # Handle printf statement
        elif next_tok == 'SCANF':
            statements.append(self.parse_scanf_statement())  # Handle scanf statement
        elif next_tok == 'IF':
            statements.append(self.parse_if_statement())  # Handle if statement
        elif next_tok == 'WHILE':
            statements.append(self.parse_while_statement())  # Handle while statement
        elif next_tok == 'RETURN':
            statements.append(self.parse_return_statement())  # Handle return statement
        else:
            raise SyntaxError(f"Unexpected token: {self.peek()}")
    return statements
```

Phần Parser được triển khai theo phương pháp đệ quy trong cùng file compiler.py, bao gồm lớp Parser với các phương thức:

- parse(): Bắt đầu phân tích, kiểm tra mã nguồn phải có cấu trúc int main() { ... }. Cụ thể gọi lần lượt:
    - consume('TYPE','int'), consume('IDENTIFIER','main'), consume('LPAREN'), consume('RPAREN'), consume('LBRACE').
    - Sau đó gọi parse_statements() để phân tích các câu lệnh bên trong { ... }, rồi consume('RBRACE') kết thúc hàm.
- parse_statements(): Lặp liên tục cho đến khi gặp token RBRACE (dấu }). Ở mỗi vòng lặp, kiểm tra token hiện tại (peek()):
    - Nếu là TYPE, gọi parse_declaration().
    - Nếu là IDENTIFIER, gọi parse_assignment().
    - Nếu là PRINTF, gọi parse_printf_statement().
    - Nếu là SCANF, gọi parse_scanf_statement().
    - Nếu là IF, gọi parse_if_statement().
    - Nếu là WHILE, gọi parse_while_statement().

- Nếu là RETURN, gọi parse_return_statement().
- Ngược lại, báo lỗi (Unexpected token).

Mỗi hàm parse_X sẽ consume các token tương ứng và trả về một nút AST (dạng dictionary) chứa thông tin của câu lệnh. Ví dụ:

- parse_declaration(): Xử lý khai báo biến. Thực hiện consume('TYPE') để lấy kiểu ('int' hoặc 'string'), consume('IDENTIFIER') lấy tên biến. Nếu gặp 'ASSIGN', parse biểu thức bên phải. Cuối cùng consume('SEMI'). Trả về nút { 'type':'declaration', 'var_type': kiểu, 'var_name': tên, 'initial_value': expr }.
- parse_assignment(): consume('IDENTIFIER') lấy tên biến trái, consume('ASSIGN'), rồi nếu token tiếp là SCANF, gọi parse_scanf_call(), ngược lại gọi parse_expression(). Cuối cùng consume('SEMI'). Trả về {'type':'assignment', 'var_name': tên, 'rhs': giá trị}.
- parse_scanf_call(): Gọi khi một biểu thức có scanf. Ví dụ trong x = scanf("%d", &y). Tiến hành consume('SCANF'), consume('LPAREN'), lấy chuỗi định dạng, consume('COMMA'), kiểm tra có AMPERSAND không để xác định is_address, lấy biến, rồi consume('RPAREN'). Trả về {'type':'scanf_call', 'format': format, 'var_name': tên, 'is_address': bool}.
- parse_printf_statement() và parse_scanf_statement(): Tương tự với cú pháp không nằm trong gán, trả về 'printf_statement' hoặc 'scanf_statement' với thông tin chuỗi định dạng và đối số (của printf) hoặc tên biến (của scanf).
- parse_if_statement(): consume('IF'), consume('LPAREN'), gọi parse_condition(), consume('RPAREN'), consume('LBRACE'), rồi gọi parse_statements() cho thân if, consume('RBRACE'). Nếu gặp ELSE, consume ELSE, {, parse thân else, }. Trả về nút {'type':'if_statement', 'condition': cond, 'if_body': [...], 'else_body': [...]}.
- parse_while_statement(): Tương tự if nhưng không có phần else. Trả về {'type':'while_statement', 'condition': cond, 'body': [...]}.

- parse_return_statement(): consume('RETURN'), parse_expression() cho biểu thức trả về, consume('SEMI'). Trả về {'type':'return_statement', 'expr': expr}.
- parse_condition(): Parse một biểu thức trái, rồi consume('COMPARE_OP'), rồi parse biểu thức phải. Trả về {'type':'condition', 'left': expr1, 'op': so_sánh, 'right': expr2}.
- Phân tích biểu thức: parse_expression(), parse_term(), parse_factor() triển khai phép toán nhị phân. Cụ thể, hiện parser hỗ trợ một phép cộng/trừ trong parse_expression và một phép nhân/chia trong parse_term, theo thứ tự ưu tiên toán học cơ bản. parse_factor() xử lý số, chuỗi, biến, hoặc biểu thức trong ngoặc (...).

Kết quả của parser là một AST (Abstract Syntax Tree), có nút gốc { 'type':'program', 'body': [...] }, trong đó 'body' là danh sách các nút câu lệnh trong main.

Chú ý: Nếu câu lệnh chỉ là a = b; (không toán tử rõ ràng), parser logic sẽ xử lý thành lhs='b', op_tok có thể sẽ lấy nhầm. Nhưng trong code, ta có xử lý nếu không thấy toán tử thì coi như giá trị 0. Thực tế, phần này parser chưa xử lý tốt (thiết kế grammar có lỗ hổng), nhưng vì tập test tập trung vào trường hợp có toán tử nên không gặp lỗi.

- Print: biểu diễn lời gọi printf. Thuộc tính:
    - fmt: chuỗi định dạng (bao gồm cả dấu nháy kép trong giá trị, ví dụ "%d").
    - arg: đối số cần in (có thể là None nếu printf không có tham số thứ hai).
- Return: biểu diễn câu lệnh return. Thuộc tính val là giá trị trả về (kiểu int).
- Input: biểu diễn lời gọi scanf. Thuộc tính name là tên biến sẽ nhận dữ liệu (bên trong mã C viết &name nhưng AST chỉ cần lưu tên biến, vì ký tự & không có trong xử lý tại codegen – codegen sẽ tự thêm lea).

Các lớp trên chủ yếu dùng như cấu trúc dữ liệu, không có phương thức phức tạp, chỉ chứa thông tin.

Biến toàn cục của parser: Parser sử dụng danh sách token do lexer tạo ra. Để tiện dùng trong nhiều hàm, ta định nghĩa:

```
# Parser globals
tokens = []
pos = 0
def peek():
    return tokens[pos] if pos < len(tokens) else (None, None)

def consume(expected=None):
    global pos
    tok = peek()
    token_types = {'KEYWORD','NUMBER','IDENT','CHAR_LIT','STR_LIT','SYMBOL','OP'}
    if expected is not None:
        if expected in token_types:
            if tok[0] != expected:
                raise SyntaxError(f"Expected token type {expected}, got {tok}")
        else:
            if tok[1] != expected:
                raise SyntaxError(f"Expected literal {expected}, got {tok}")
    pos += 1
    return tok
```

trong đó tokens sẽ được gán bằng list token đầu vào, còn pos là chỉ số hiện tại trong list (giả lập con trỏ đọc token). Có hai hàm hỗ trợ:

- peek() – xem token hiện tại (ở vị trí pos) nhưng không tiến pos. Nếu đã hết token thì trả (None, None).
- consume(expected=None) – lấy token hiện tại rồi tăng pos lên 1. Tham số expected cho phép đặt kỳ vọng: nếu expected không None, hàm sẽ kiểm tra token hiện tại xem có khớp kỳ vọng không:
    - Nếu expected là một loại token (KEYWORD, IDENT, ...), nó kiểm tra tok[0] (loại) có đúng không.
    - Nếu expected là một giá trị cụ thể (chuỗi ký tự như {, ;), nó kiểm tra tok[1] (giá trị chuỗi) khớp không.
    - Nếu không khớp, sẽ ném SyntaxError báo thông tin, dừng parser.
    - Nếu khớp hoặc expected là None (không kiểm tra), thì trả về token hiện tại và tiến pos.

Hàm consume rất quan trọng để đảm bảo dòng token đúng cú pháp mong đợi. Mỗi khi parser "mong đợi" một token nào đó (ví dụ một dấu { hay một từ khoá), nó sẽ gọi consume với tham số tương ứng. Nếu token không như kỳ vọng, parser báo lỗi ngay.

Hàm chính parse_program():

```
# Parser functions
def parse_program():
    global pos
    pos = 0
    prog = Program()
    consume('int'); consume('main'); consume('('); consume(')'); consume('{')
    while peek()[1] in ('int', 'char'):
        consume()  # type
        name = consume('IDENT')[1]
        if peek()[1] == '=':
            consume('='); val = consume()[1]
        else:
            val = '0'
        consume(';')
        prog.decls.append(Decl(name, val))
    while peek()[1] != '}':
        tok = peek()[1]
        if tok == 'if':
            prog.stmts.append(parse_if())
        elif tok == 'printf':
            prog.stmts.append(parse_printf(prog))
        elif tok == 'scanf':
            prog.stmts.append(parse_scanf(prog))
        elif tok == 'return':
            prog.stmts.append(parse_return())
        else:
            prog.stmts.append(parse_assign())
    consume('}')
    return prog
```

Giải thích:

- Đặt pos = 0 để bắt đầu từ đầu danh sách token.
- Khởi tạo một đối tượng Program để chứa kết quả parse.
- Trước tiên, consume các token bắt buộc mở đầu: phải gặp int, rồi main, rồi (,), {. Nếu bất kỳ cái nào không đúng (ví dụ thiếu int đầu), consume sẽ báo lỗi. Điều này tương ứng việc parser *expect* mã nguồn bắt đầu bằng int main() {. Chúng ta bỏ qua giá trị 'int' và 'main' (không lưu trong AST, vì AST đã ngầm định Program tương ứng main function).

- Sau dấu {, parser mong đợi có thể là các khai báo biến. Theo ngữ pháp, parser sẽ đọc tuần tự các token:
    - Nếu token kế tiếp là 'int' hoặc 'char' (một kiểu dữ liệu), thì đó là bắt đầu một khai báo. Thực hiện vòng while:
        - consume() lần đầu tiên lấy token kiểu (int/char) nhưng không lưu nó (vì hiện tại ta không thật sự dùng thông tin kiểu khác biệt, cả hai đều thành biến trong .data).
        - Sau đó, consume('IDENT') để lấy tên biến, trả về token, lấy phần [1] (giá trị chuỗi) lưu vào name.
        - Kiểm tra peek()[1] xem có phải '=' không (tức là có khởi tạo không):
            - Nếu có '=' thì consume nó và consume token tiếp theo làm giá trị khởi tạo (val).
            - Nếu không có, đặt val = '0' (string '0').
        - consume(';') để đảm bảo kết thúc khai báo bằng dấu chấm phẩy.
        - Tạo đối tượng Decl(name, val) và append vào prog.decls.
    - Vòng lặp này tiếp tục cho đến khi không còn thấy kiểu dữ liệu int/char nữa, nghĩa là khai báo kết thúc.
- Kế tiếp, parser đọc các token tiếp theo cho phần thân hàm (các câu lệnh). Nó lặp while peek()[1] != '}' – tức là đọc đến khi gặp dấu } kết thúc hàm main:
    - Lấy tok = peek()[1] (giá trị của token hiện tại, loại thứ [0] không cần thiết ở đây vì ta phân biệt dựa trên giá trị cụ thể).
    - Nếu tok == 'if': gặp từ khóa if, gọi hàm parse_if() để xử lý cả cấu trúc if…else, kết quả trả về một đối tượng If thêm vào prog.stmts.
    - Elif tok == 'printf': gặp hàm in, gọi parse_printf(prog). Lưu ý ta truyền prog vào, vì trong khi parse printf, ta sẽ cần lưu literal chuỗi định dạng vào danh sách prog.strings. Kết quả trả về Print node thêm vào stmts.

- Elif tok == 'scanf': tương tự, gọi parse_scanf(prog) để xử lý, node trả về kiểu Input.
- Elif tok == 'return': gọi parse_return(), trả về node Return.
- Else: Nếu token không phải bốn trường hợp trên, ta mặc định đó phải là một lệnh gán (hoặc một biểu thức dạng gán). Gọi parse_assign().

- Cuối cùng, consume('}') để đọc dấu } kết thúc hàm main. Nếu thiếu }, parser sẽ lỗi. Trả về đối tượng Program đã điền đủ decls và stmts.

Hàm parse_if():

```
def parse_if():
    consume('if'); consume('(')
    lhs = consume()[1]
    op  = consume()[1]
    rhs = consume()[1]
    consume(')'); consume('{')
    then_stmts = []
    while peek()[1] != '}': then_stmts.append(parse_assign())
    consume('}')
    else_stmts = []
    if peek()[1] == 'else':
        consume('else'); consume('{')
        while peek()[1] != '}': else_stmts.append(parse_assign())
        consume('}')
    return If(Cond(lhs, op, rhs), then_stmts, else_stmts)
```

Giải thích:

- Bắt đầu đã biết token hiện tại là 'if' (đảm bảo bởi gọi từ parse_program), consume nó. Sau đó consume dấu '('.
- Bên trong (), parser **giản lược**: nó không gọi một hàm parse biểu thức riêng, mà tự làm:
  - lhs = consume()[1] – lấy token kế tiếp (bất kể loại gì) làm vế trái (có thể là IDENT hoặc NUMBER).
  - op = consume()[1] – lấy token kế tiếp làm toán tử (theo token spec thì chắc chắn loại OP, ví dụ '>' hoặc '==' v.v.).
  - rhs = consume()[1] – lấy tiếp token làm vế phải (IDENT hoặc NUMBER).
  - Cách làm này hiệu quả trong trường hợp điều kiện đúng chuẩn dạng <expr> <op> <expr> đơn giản. Nếu mã nguồn khác format (ví dụ if (x), thiếu toán tử và

rhs) thì parser vẫn cố lấy 3 token => sẽ sai (có thể gây IndexError hoặc token bất ngờ).

- Sau khi lấy đủ 3 thành phần điều kiện, consume) và { để bắt đầu then-block.
- then_stmts: khởi tạo list rỗng, sau đó while peek()[1] != '}': khi chưa gặp }, parse các câu lệnh bên trong. Theo thiết kế, chỉ có các câu lệnh gán nên ta gọi parse_assign() cho từng cái. (Nếu code có thứ khác, ví dụ một printf trong if, thì peek()[1] sẽ là "printf" != '}' nên nó sẽ đi vào parse_assign(), parse_assign thấy token đầu không phải IDENT sẽ lỗi. Tức là parser không cho phép, như phân tích ở trên).
- consume '}' để kết thúc then-block.
- else_stmts: kiểm tra nếu token tiếp theo là 'else':
  - Nếu có: consume 'else', consume '{', rồi trong khi chưa gặp '}', parse_assign() cho từng lệnh, sau đó consume '}'.
  - Nếu không có 'else', else_stmts giữ list rỗng.
- Cuối cùng, trả về một đối tượng If với Cond và 2 danh sách. Ở đây, Cond được tạo bằng Cond(lhs, op, rhs) với lhs, op, rhs đều là chuỗi (ví dụ lhs='a', op='>', rhs='10').

Hàm parse_assign():

```
def parse_assign():
    name = consume('IDENT')[1]
    consume('OP')
    lhs = consume()[1]
    op_tok = consume('OP')[1]
    rhs = consume()[1]
    consume(';')
    return Assign(name, lhs, op_tok, rhs)
```

Cách parse assignment này còn khá đơn giản

- Đầu tiên, nó mong đợi một token IDENT (tên biến) và lấy tên đó làm name.
- Sau đó consume('OP') – ở đây, kỳ vọng token tiếp theo phải thuộc loại OP. Lý do: dấu = trong token list được phân loại là loại OP (theo TOK_SPEC, '=' là OP). Cho nên consume('OP') thực chất đang tiêu thụ dấu bằng. **Nhược điểm**: nếu vì lý do gì đó token

= không được lexer coi là OP (nhưng thực tế có), hoặc nếu người dùng viết code sai (ví dụ a + b; thiếu =), parser sẽ cố consume('OP') và có thể lấy nhầm token '+' (loại OP) thay vì '='. Bởi code không kiểm tra giá trị token ở đây. Đáng ra nên viết consume('=') thay vì consume('OP') để chắc chắn. Đây có thể coi là một lỗi nhỏ trong triển khai parser, nhưng giả sử người dùng viết đúng cú pháp thì token tại đó chắc chắn là '=' nên vẫn hoạt động.

- Sau khi consume '=' (loại OP), nó lấy tiếp token làm lhs (không kiểm tra loại, có thể là IDENT, NUMBER hoặc CHAR_LIT).
- Rồi consume('OP') lần nữa để lấy toán tử. Lần này, do ngay trước đó ta lấy ví dụ lhs, token kế tiếp của biểu thức gán phải là toán tử nhị phân (+, -, *, /). Consume('OP') sẽ lấy token đó, trả về [1] (ví dụ '+'), lưu vào op_tok.
- Kế tiếp, rhs = consume()[1] lấy toán hạng bên phải (có thể là số hoặc biến).
- Cuối cùng, consume(';' ký tự) để kết thúc câu lệnh.
- Trả về Assign(name, lhs, op_tok, rhs).

Như vậy parse_assign xử lý trường hợp câu lệnh gán có dạng tổng quát: <name> = <lhs> <op> <rhs> ;. Ví dụ a = b + 5; sẽ cho: name='a', lhs='b', op_tok='+', rhs='5'. Tuy nhiên, có vấn đề nếu câu lệnh chỉ là a = b; (không có). Giả sử token là a, '=' , b, ';' – parser:

- name='a'
- consume('OP') lấy '='
- lhs = consume()[1] lấy 'b'
- op_tok = consume('OP')[1] — ở đây, token kế tiếp là ';' (loại SYMBOL). consume('OP') sẽ lỗi vì ';' không phải loại OP. Điều này cho thấy grammar parser viết ở đây không xử lý câu lệnh gán đơn giản không có toán tử. Nếu muốn gán trực tiếp, có thể coi là + 0 hoặc * 1. Quả thực, trong phần code tạo AST, nếu không có toán tử thì lẽ ra parse_assign nên làm trường hợp else. Nhưng do không có, chương trình sẽ lỗi nếu gặp a = b;. Để tránh lỗi này, trong tập sử dụng, viết a = b + 0; thay vì a = b;. Hoặc có thể fix parser cho robust hơn. Đây là một giới hạn cần lưu ý khi sử dụng compiler. (Một

cách khác: trình biên dịch có thể thêm logic: nếu token sau lhs là ';' thì ngầm hiểu op là '+' và rhs là '0'. Nhưng code hiện tại chưa làm điều đó).

Hàm parse_printf(prog):

```
def parse_printf(prog):
    consume('printf')
    consume('(')
    fmt_tok = consume('STR_LIT')[1]
    prog.strings.append(fmt_tok)
    arg = None
    if peek()[1] == ',':
        consume(',')
        arg_tok = peek()
        if arg_tok[0] in ('IDENT', 'NUMBER', 'CHAR_LIT'):
            arg = consume()[1]
        else:
            raise SyntaxError(f"Expected IDENT, NUMBER, or CHAR_LIT, got {arg_tok}")
    consume(')')
    consume(';')
    return Print(fmt_tok, arg)
```

Giải thích:

- Consume từ khóa "printf" và dấu (.
- Tiếp theo, kỳ vọng một chuỗi định dạng: consume('STR_LIT'). Lấy giá trị chuỗi (bao gồm cả dấu ngoặc kép) đưa vào fmt_tok.
- Thêm chuỗi định dạng này vào prog.strings. Mục đích: tất cả chuỗi (của cả printf và scanf) được gom vào danh sách prog.strings để codegen sẽ tạo dữ liệu cho chúng trong section .data với các nhãn riêng (str_0, str_1, ...).
- Thiết lập arg = None. Sau đó kiểm tra peek()[1]: nếu là dấu phẩy , nghĩa là có tham số thứ hai:
    - consume dấu ,, rồi xem token kế tiếp (arg_tok = peek()). Nếu loại token của nó thuộc một trong IDENT, NUMBER, CHAR_LIT, thì chấp nhận: gán arg = consume()[1] (lấy giá trị chuỗi của token đó). Nếu token tiếp theo không phải biến, số hoặc ký tự (ví dụ nhỡ viết một biểu thức phức tạp hay hàm gì đó), parser sẽ raise SyntaxError "Expected IDENT, NUMBER or CHAR_LIT".
- Sau đó consume) và ; kết thúc.
- Trả về node Print(fmt_tok, arg).

Như vậy, parse_printf đảm bảo cú pháp printf("format", optional_arg); với optional_arg nếu có phải là một biến/hằng/char. Nó không cho phép biểu thức hay gọi hàm bên trong printf (theo mục tiêu đơn giản).

Hàm parse_scanf(prog):

```
def parse_scanf(prog):
    consume('scanf')
    consume('(')
    fmt_tok = consume('STR_LIT')[1]
    prog.strings.append(fmt_tok)
    consume(',')
    amp = consume('OP')
    if amp[1] != '&':
        raise SyntaxError("Expected '&' before variable in scanf")
    name = consume('IDENT')[1]
    consume(')')
    consume(';')
    return Input(name)
```

Giải thích:

- Tương tự printf, consume "scanf", "(".
- Đọc chuỗi định dạng fmt_tok (kỳ vọng STR_LIT). Thêm fmt_tok vào prog.strings.
- Consume dấu phẩy.
- Sau dấu phẩy, kỳ vọng ký tự &. Ở lexer, & được phân loại là loại OP. Ta gọi amp = consume('OP'). Sau đó kiểm tra amp[1] (giá trị chuỗi) có phải '&' không. Nếu không, parser báo lỗi yêu cầu phải có & trước biến (theo cú pháp C).
- Nếu đúng, consume token kế tiếp phải là IDENT (tên biến nhận dữ liệu) -> lưu vào name.
- Consume) và ; kết thúc.
- Trả về node Input(name).

Cũng như printf, parse_scanf đơn giản chấp nhận một biến sau dấu &. Nó không cho phép đọc nhiều biến cùng lúc (chỉ một biến) và giả định format chỉ có một %d. Tương tự, do ta append chuỗi format vào prog.strings, codegen sẽ tạo một str_x cho "%d" này, mặc dù trong codegen ta lại dùng một nhãn chung fmt cho scanf (chi tiết ở phần sau).

Hàm parse_return():

```
def parse_return():
    consume('return')
    val_tok = consume()
    val = val_tok[1] if val_tok[0] == 'NUMBER' else '0'
    consume(';')
    return Return(int(val))
```

Giải thích:

- Consume "return".
- Lấy token kế tiếp (có thể là số hoặc có thể là ; ngay nếu người dùng viết return; không giá trị). Lưu token vào val_tok.
- Nếu val_tok[0] == 'NUMBER' (loại number) thì lấy giá trị chuỗi, ngược lại (nếu không phải number, ví dụ trường hợp return x; hoặc return;) thì đặt '0'. Cách làm này nghĩa là parser chỉ chính thức hỗ trợ return số. Nếu return x; (x là biến), val_tok[0] sẽ là 'IDENT', không phải 'NUMBER', do đó nó sẽ đặt val = '0'. Tức là **bỏ qua giá trị trả về người lập trình muốn** – một hạn chế nữa. Có lẽ tác giả chỉ mong dùng return 0; trong main. Bởi return các giá trị khác trong main cũng không quá quan trọng.
- Dù sao, nó convert val sang int (vì Return node expects an int in val attribute).
- Consume ;.
- Trả về node Return(int(val)).

Tóm lại, parser của chúng ta có nhiều chỗ giản lược theo hướng "hard-code" logic cho các trường hợp đơn giản, thay vì xây dựng grammar tổng quát. Điều này giúp code ngắn gọn nhưng đánh đổi tính tổng quát. Tuy nhiên, cho các trường hợp hợp lệ mà chúng ta dự kiến (theo tập con C), parser vẫn hoạt động đúng và tạo ra AST mong đợi.

4.3.3. Phân Tích Ngữ Nghĩa

Semantic Analyzer (bộ phân tích ngữ nghĩa) kiểm tra tính hợp lệ logic của chương trình, đảm bảo rằng mã nguồn không chỉ đúng cú pháp mà còn có ý nghĩa về mặt ngữ nghĩa. Các nhiệm vụ chính bao gồm kiểm tra khai báo biến, phát hiện trùng lặp tên biến, và đảm bảo các biểu thức sử dụng biến đã được khai báo.

```
class SemanticAnalyzer:
    def __init__(self, ast):
        self.symbol_table = {}
        self.ast = ast

    def check(self):
        self.visit_node(self.ast)

    def visit_node(self, node):
        if node['type'] == 'program':
            for stmt in node['body']:
                self.visit_node(stmt)
        elif node['type'] == 'declaration':
            var_name = node['var_name']
            if var_name in self.symbol_table:
                raise NameError(f"Duplicate declaration: {var_name}")
            self.symbol_table[var_name] = node['var_type']
            if node['initial_value']:  # Kiểm tra nếu có giá trị khởi tạo
                self.visit_expression(node['initial_value'])
        elif node['type'] == 'assignment':
            var_name = node['var_name']
            if var_name not in self.symbol_table:
                raise NameError(f"Undeclared variable: {var_name}")
            self.visit_expression(node['rhs'])
        elif node['type'] == 'printf_statement':
            for arg in node['args']:
                self.visit_expression(arg)
        elif node['type'] == 'scanf_statement':
            var_name = node['var_name']
            if var_name not in self.symbol_table:
                raise NameError(f"Undeclared variable: {var_name}")
        elif node['type'] == 'if_statement':
            self.visit_condition(node['condition'])
            for stmt in node['if_body']:
                self.visit_node(stmt)
            for stmt in node.get('else_body', []):
                self.visit_node(stmt)
```

```
        elif node['type'] == 'while_statement':
            self.visit_condition(node['condition'])
            for stmt in node['body']:
                self.visit_node(stmt)
        elif node['type'] == 'return_statement':
            self.visit_expression(node['expr'])

    def visit_expression(self, node):
        if node['type'] == 'identifier':
            if node['name'] not in self.symbol_table:
                raise NameError(f"Undeclared variable: {node['name']}")
        elif node['type'] in ['number', 'string_literal']:
            pass  # Không cần kiểm tra thêm cho number hoặc string_literal
        elif node['type'] == 'binary_op':
            self.visit_expression(node['left'])
            self.visit_expression(node['right'])
        else:
            raise ValueError(f"Unknown expression type: {node['type']}")

    def visit_condition(self, node):
        self.visit_expression(node['left'])
        self.visit_expression(node['right'])
```

Sau khi có AST, bước tiếp theo là kiểm tra ngữ nghĩa bằng lớp SemanticAnalyzer:

- **Bảng ký hiệu (symbol table):** Được lưu dưới dạng dictionary, với khóa là tên biến, giá trị là kiểu (int hoặc string) và có thể là kích thước.
- visit_node(node): Duyệt đệ quy các nút AST:
    - Với type='program', lần lượt duyệt các câu lệnh trong 'body'.
    - Với type='declaration', lấy var_name, kiểm tra chưa khai báo trước đó (nếu có báo lỗi Duplicate declaration), sau đó thêm vào symbol_table. Nếu có initial_value, gọi visit_expression() với giá trị khởi tạo.
    - Với type='assignment', kiểm tra var_name đã khai báo chưa (nếu chưa có, báo Undeclared variable), sau đó visit_expression(rhs).
    - Với printf_statement, với mỗi đối số trong 'args', gọi visit_expression(arg).
    - Với scanf_statement, kiểm tra var_name đã khai báo (yêu cầu nhập biến đã khai báo).
    - Với if_statement, gọi visit_condition(condition), sau đó duyệt các câu lệnh trong if_body và else_body (nếu có).
    - Với while_statement, tương tự với condition và body.
    - Với return_statement, gọi visit_expression(expr).
- visit_expression(node): Kiểm tra biểu thức:
    - Nếu là identifier, kiểm tra biến đã khai báo trong symbol_table.
    - Nếu là number hoặc string_literal, chấp nhận.
    - Nếu là binary_op, đệ quy kiểm tra left và right.
- visit_condition(condition): Đệ quy kiểm tra hai bên left và right giống như biểu thức.

Nếu phát hiện bất kỳ lỗi ngữ nghĩa nào (biến chưa khai báo, khai báo trùng), chương trình ném ngoại lệ và dừng biên dịch. Nếu không có lỗi, Semantic Analyzer kết thúc, cho phép tiếp tục bước tạo mã.

Ý nghĩa thực tiễn: Semantic Analyzer đóng vai trò như một lớp bảo vệ, đảm bảo rằng chương trình không chỉ đúng cú pháp mà còn có thể thực thi mà không gặp lỗi logic. Bảng ký hiệu đơn giản nhưng hiệu quả trong việc quản lý thông tin biến, đặc biệt với tập con C nhỏ. Tuy nhiên, hệ thống hiện tại chưa kiểm tra tính tương thích kiểu dữ liệu (ví dụ: gán chuỗi cho biến int), điều này có thể được cải thiện trong tương lai.

4.3.4. Sinh mã hợp ngữ (Code generation)

Bộ Sinh Mã Hợp Ngữ (Code Generator) là thành phần cuối cùng trong chuỗi xử lý của trình biên dịch, chịu trách nhiệm chuyển đổi cây cú pháp trừu tượng (AST) thành mã hợp ngữ x86-64, tạo ra một chương trình thực thi được. Đây là bước kết nối mã nguồn cấp cao (ngôn ngữ C) với phần cứng, đảm bảo rằng các câu lệnh C được ánh xạ chính xác thành các lệnh máy có thể chạy trực tiếp trên CPU.

Tính chính xác: Code Generator phải đảm bảo rằng mỗi node trong AST được ánh xạ thành một chuỗi lệnh hợp ngữ phản ánh đúng ý nghĩa ngữ nghĩa của câu lệnh C. Ví dụ, một phép gán như x = y + 5; phải tạo ra các lệnh tải giá trị, thực hiện phép cộng, và lưu kết quả.

Hiệu suất cơ bản: Việc sử dụng thanh ghi (eax, ebx, rdi, rsi) để xử lý các phép toán và tham số hàm giúp giảm thời gian truy cập bộ nhớ, tăng hiệu suất so với việc chỉ sử dụng ngăn xếp.

Tính tương thích: Mã hợp ngữ được sinh ra phải tương thích với trình hợp dịch GCC, cho phép liên kết với thư viện chuẩn C (như printf và scanf) để tạo tệp thực thi hoàn chỉnh.

Tính mô-đun: Code Generator được thiết kế để xử lý từng loại node AST riêng lẻ (như declaration, if_statement, printf_statement), giúp dễ dàng mở rộng để hỗ trợ các cấu trúc mới như for hoặc hàm con trong tương lai.

Triển khai: Trong file compiler.py, lớp CodeGenerator được triển khai để duyệt AST và sinh mã hợp ngữ. Lớp này sử dụng các kỹ thuật quản lý ngăn xếp để lưu trữ biến và thanh ghi để thực hiện các phép toán, đồng thời tạo các nhãn (label) để điều khiển luồng chương trình. Dưới đây, tôi sẽ phân tích chi tiết từng phương thức và đoạn mã quan trọng, giải thích chức năng chính, và cung cấp ví dụ minh họa.

Khởi Tạo (__init__) :

```
class CodeGenerator:
    def __init__(self):
        self.code = []
        self.label_count = 0
        self.symbol_table = {}
        self.current_offset = -8
        self.stack_size = 0  # Theo dõi kích thước stack
```

Khởi tạo các thuộc tính cần thiết để sinh mã hợp ngữ:

- code: Danh sách chứa các dòng lệnh hợp ngữ được sinh ra, sẽ được nối thành chuỗi cuối cùng.
- label_count: Bộ đếm để tạo các nhãn duy nhất (như .L1, .L2) cho các lệnh nhảy trong if và while.
- symbol_table: Từ điển lưu thông tin về biến (offset trên ngăn xếp, kiểu dữ liệu, kích thước), được sử dụng để ánh xạ tên biến sang vị trí bộ nhớ.
- current_offset: Theo dõi vị trí hiện tại trên ngăn xếp, bắt đầu từ -8 và giảm dần để cấp phát không gian cho biến.
- stack_size: Theo dõi tổng kích thước ngăn xếp được cấp phát, đặc biệt quan trọng với biến kiểu string (256 byte).

Ý nghĩa: Việc khởi tạo các thuộc tính này đảm bảo rằng Code Generator có thể quản lý trạng thái của chương trình (biến, nhãn, ngăn xếp) một cách nhất quán, tránh xung đột hoặc truy cập bộ nhớ không hợp lệ.

Phương Thức emit :

```
    def emit(self, line):
        self.code.append(line)
```

Thêm một dòng lệnh hợp ngữ vào danh sách code. Đây là phương thức cơ bản để ghi lại các lệnh được sinh ra, từ các lệnh khởi tạo ngăn xếp (pushq %rbp) đến các lệnh gọi hàm (call printf).

Ý nghĩa: Phương thức này đảm bảo rằng các lệnh hợp ngữ được ghi lại theo thứ tự chính xác, tạo ra một chuỗi mã hoàn chỉnh có thể lưu vào tệp .s. Nó đơn giản nhưng thiết yếu, đóng vai trò như giao diện giữa logic sinh mã và đầu ra.

Phương Thức generate :

```
def generate(self, ast):
    self.emit('.section .text')
    self.emit('.globl main')
    self.emit('main:')
    self.emit('pushq %rbp')
    self.emit('movq %rsp, %rbp')
    self.allocate_variables(ast)
    for stmt in ast['body']:
        self.generate_stmt(stmt)
    self.emit('movq $0, %rax')
    self.emit('leave')
    self.emit('ret')
    return '\n'.join(self.code)
```

Sinh mã hợp ngữ cho toàn bộ chương trình, bắt đầu bằng các lệnh thiết lập và kết thúc bằng các lệnh dọn dẹp:

- .section .text: Đặt mã hợp ngữ trong phân đoạn mã (text section).
- .globl main: Khai báo hàm main là toàn cục, cho phép liên kết với thư viện C.
- pushq %rbp, movq %rsp, %rbp: Thiết lập khung ngăn xếp (stack frame) cho hàm main.
- allocate_variables(ast): Cấp phát không gian trên ngăn xếp cho các biến.
- generate_stmt(stmt): Sinh mã cho từng câu lệnh trong thân chương trình.
- movq $0, %rax, leave, ret: Đặt giá trị trả về là 0, dọn dẹp ngăn xếp, và thoát khỏi hàm main.

Ý nghĩa: Phương thức này là điểm nhập chính của Code Generator, điều phối toàn bộ quá trình sinh mã. Nó đảm bảo rằng mã hợp ngữ tuân theo quy ước gọi hàm của x86-64 (System V ABI), cho phép tích hợp với các hàm thư viện như printf.

Phương Thức allocate_variables

```
def allocate_variables(self, ast):
    for node in ast['body']:
        if node['type'] == 'declaration':
            var_name = node['var_name']
            var_type = node['var_type']
            # Cấp phát offset cho biến
            if var_type == 'string' and not node['initial_value']:
                # Cấp phát 256 byte cho string trên stack
                self.stack_size += 256
                self.symbol_table[var_name] = {'offset': self.current_offset, 'type': var_type, 'size': 256}
                self.emit(f'subq $256, %rsp')
                self.emit(f'movq %rsp, {self.current_offset}(%rbp)')
            else:
                self.symbol_table[var_name] = {'offset': self.current_offset, 'type': var_type, 'size': 8}
                self.emit(f'movl $0, {self.current_offset}(%rbp)')  # Khởi tạo mặc định
            self.current_offset -= 8

            # Xử lý initial_value
            if node['initial_value']:
                if var_type == 'int':
                    if node['initial_value']['type'] == 'number':
                        self.emit(f'movl ${node["initial_value"]["value"]}, {self.symbol_table[var_name]["offset"]}(%rbp)')
                    elif node['initial_value']['type'] == 'binary_op':
                        left = node['initial_value']['left']
                        right = node['initial_value']['right']
                        op = node['initial_value']['op']
                        if left['type'] == 'identifier':
                            self.emit(f'movl {self.symbol_table[left["name"]]["offset"]}(%rbp), %eax')
                        elif left['type'] == 'number':
                            self.emit(f'movl ${left["value"]}, %eax')
                        if right['type'] == 'identifier':
                            self.emit(f'movl {self.symbol_table[right["name"]]["offset"]}(%rbp), %ebx')
                        elif right['type'] == 'number':
                            self.emit(f'movl ${right["value"]}, %ebx')
```

```
                        if op == '+':
                            self.emit('addl %ebx, %eax')
                        elif op == '-':
                            self.emit('subl %ebx, %eax')
                        elif op == '*':
                            self.emit('imull %ebx, %eax')
                        elif op == '/':
                            self.emit('cdq')
                            self.emit('idivl %ebx')
                        self.emit(f'movl %eax, {self.symbol_table[var_name]["offset"]}(%rbp)')
                elif var_type == 'string':
                    label = self.new_label()
                    self.emit('.section .rodata')
                    self.emit(f'{label}: .string {node["initial_value"]["value"]}')
                    self.emit('.section .text')
                    self.emit(f'lea {label}(%rip), %rax')
                    self.emit(f'movq %rax, {self.symbol_table[var_name]["offset"]}(%rbp)')
```

Duyệt qua AST để tìm các node declaration và cấp phát không gian trên ngăn xếp cho các biến:

- Với biến kiểu string không có giá trị khởi tạo, cấp phát 256 byte và lưu con trỏ ngăn xếp.
- Với biến kiểu int hoặc string có giá trị khởi tạo, cấp phát 8 byte và khởi tạo giá trị mặc định là 0.
- Lưu thông tin biến (offset, kiểu, kích thước) vào symbol_table.

Xử lý giá trị khởi tạo:

- Với int, hỗ trợ số (number) hoặc biểu thức số học (binary_op), sử dụng thanh ghi eax, ebx để tính toán.
- Với string, tạo nhãn trong phân đoạn .rodata để lưu chuỗi và lưu địa chỉ chuỗi vào ngăn xếp.

Ý nghĩa: Phương thức này đảm bảo rằng mỗi biến được cấp phát không gian bộ nhớ phù hợp và giá trị khởi tạo được xử lý chính xác, tạo nền tảng cho các câu lệnh tiếp theo sử dụng biến.

Phương Thức generate_stmt :

```
def generate_stmt(self, node):
    if node['type'] == 'printf_statement':
        self.generate_printf(node)
    elif node['type'] == 'scanf_statement':
        self.generate_scanf(node)
    elif node['type'] == 'assignment':
        self.generate_assignment(node)
    elif node['type'] == 'if_statement':
        self.generate_if(node)
    elif node['type'] == 'while_statement':
        self.generate_while(node)
    elif node['type'] == 'return_statement':
        self.generate_return(node)
```

Điều phối việc sinh mã cho từng loại câu lệnh trong AST bằng cách gọi các phương thức chuyên biệt (generate_printf, generate_if, v.v.).

Ý nghĩa: Phương thức này đóng vai trò như một bộ định tuyến, đảm bảo rằng mỗi loại node AST được xử lý bởi logic phù hợp, tăng tính mô-đun và dễ bảo trì.

Phương Thức generate_printf :

```
def generate_printf(self, node):
    fmt_label = self.new_label()
    self.emit('.section .rodata')
    self.emit(f'{fmt_label}: .string {node["format"]}')
    self.emit('.section .text')
    self.emit(f'lea {fmt_label}(%rip), %rdi')
    if node['args']:
        arg = node['args'][0]
        if arg['type'] == 'identifier':
            var_info = self.symbol_table[arg['name']]
            if var_info['type'] == 'int':
                self.emit(f'movl {var_info["offset"]}(%rbp), %esi')
            elif var_info['type'] == 'string':
                self.emit(f'movq {var_info["offset"]}(%rbp), %rsi')
        elif arg['type'] == 'number':
            self.emit(f'movl ${arg["value"]}, %esi')
    self.emit('mov $0, %eax')
    self.emit('call printf')
```

Sinh mã hợp ngữ để gọi hàm printf:

- Tạo nhãn trong .rodata để lưu chuỗi định dạng (ví dụ: "Value: %d\n").
- Tải địa chỉ chuỗi vào thanh ghi rdi (tham số đầu tiên theo System V ABI).
- Nếu có tham số (ví dụ: biến x hoặc số 10), tải giá trị vào rsi (tham số thứ hai).
- Đặt eax = 0 (không sử dụng đối số dấu phẩy động) và gọi printf.

Ý nghĩa: Phương thức này đảm bảo rằng các lệnh printf trong mã C được ánh xạ chính xác thành các lệnh gọi hàm thư viện C, hỗ trợ xuất dữ liệu ra màn hình.

Phương Thức generate_assignment

```
def generate_assignment(self, node):
    var_name = node['var_name']
    var_offset = self.symbol_table[var_name]['offset']
    rhs = node['rhs']
    if rhs['type'] == 'binary_op':
        left = rhs['left']
        right = rhs['right']
        op = rhs['op']
        if left['type'] == 'identifier':
            self.emit(f'movl {self.symbol_table[left["name"]]["offset"]}(%rbp), %eax')
        elif left['type'] == 'number':
            self.emit(f'movl ${left["value"]}, %eax')
        if right['type'] == 'identifier':
            self.emit(f'movl {self.symbol_table[right["name"]]["offset"]}(%rbp), %ebx')
        elif right['type'] == 'number':
            self.emit(f'movl ${right["value"]}, %ebx')
        if op == '+':
            self.emit('addl %ebx, %eax')
        elif op == '-':
            self.emit('subl %ebx, %eax')
        elif op == '*':
            self.emit('imull %ebx, %eax')
        elif op == '/':
            self.emit('cdq')
            self.emit('idivl %ebx')
        self.emit(f'movl %eax, {var_offset}(%rbp)')
```

Sinh mã cho câu lệnh gán (x = ...), tập trung vào xử lý biểu thức số học (binary_op):

- Tải giá trị bên trái (left) vào eax (từ biến hoặc số).
- Tải giá trị bên phải (right) vào ebx.
- Thực hiện phép toán (addl, subl, imull, idivl) và lưu kết quả vào eax.
- Lưu kết quả từ eax vào vị trí của biến đích trên ngăn xếp.

Ý nghĩa: Phương thức này hỗ trợ các phép gán với biểu thức số học phức tạp, đảm bảo rằng các phép toán được thực hiện đúng thứ tự và kết quả được lưu trữ chính xác.

Phương Thức generate_while

```
def generate_while(self, node):
    start_label = self.new_label()
    end_label = self.new_label()
    self.emit(f'{start_label}:')
    self.generate_condition(node['condition'], end_label)
    for stmt in node['body']:
        self.generate_stmt(stmt)
    self.emit(f'jmp {start_label}')
    self.emit(f'{end_label}:')
```

Sinh mã cho vòng lặp while:

- Tạo nhãn start_label để đánh dấu điểm bắt đầu vòng lặp.
- Sinh mã kiểm tra điều kiện, nhảy đến end_label nếu điều kiện sai.
- Sinh mã cho thân vòng lặp, sau đó nhảy lại start_label để lặp lại.

Ý nghĩa: Phương thức này ánh xạ chính xác cấu trúc lặp của while thành các lệnh nhảy, đảm bảo rằng vòng lặp được thực thi đúng số lần dựa trên điều kiện.

4.4. Tổ chức mã nguồn và cách sử dụng

- **Tổ chức mã nguồn**

Mã nguồn của hệ thống được tổ chức thành các file Python và file C như sau:

compiler.py:

Chứa toàn bộ các thành phần chính của trình biên dịch, bao gồm lexer, parser, semantic analyzer, và code generator. Đây là file quan trọng nhất. File có thể chạy trực tiếp từ dòng lệnh, nhận hai đối số: tên file C input và tên file Assembly output.

Khi chạy, compiler.py sẽ thực hiện phân tích mã nguồn C và sinh ra file Assembly .s tương ứng.

code_editer.py:

Chứa **giao diện người dùng đồ họa (GUI)** cho phép:

- Soạn thảo mã nguồn C.
- Lưu file.
- Biên dịch mã C bằng cách gọi compiler.py.
- Biên dịch file Assembly sang thực thi bằng GCC.
- Thực thi chương trình trực tiếp từ giao diện. Giao diện giúp người dùng thao tác thuận tiện, không cần dùng dòng lệnh thủ công.

program.c:

Là **file mã nguồn C mẫu** cần biên dịch.Chương trình mẫu sử dụng đầy đủ các tính năng cơ bản: khai báo biến, nhập xuất (printf, scanf), phép toán số học, câu lệnh điều kiện if-else, vòng lặp while, và lệnh return.

Quá trình sử dụng hệ thống gồm các bước:

1. Người dùng soạn hoặc mở mã nguồn C trong trình chỉnh sửa GUI.
2. Khi bấm **Run Code**, chương trình lưu mã vào file tạm .c và gọi compiler.py (một script Python) để thực hiện lex, parse, semantic và sinh mã hợp ngữ (.s).
3. Nếu compiler.py trả về lỗi, thông báo lỗi được hiển thị trên khu vực console của GUI.
4. Nếu thành công, mã hợp ngữ (.s) được hiển thị ở khung console.
5. Sau đó, GUI gọi gcc -no-pie temp.s -o temp.out để biên dịch mã assembly thành file thực thi, rồi chạy file này trong một terminal (Gnome Terminal) mới. Kết quả đầu ra của chương trình hiển thị trong terminal.

Tóm tắt dòng chảy dữ liệu:

Soạn mã C → Code Editor (GUI) → Lưu file .c → **Compiler (compiler.py)** → Tạo file assembly .s → **GCC** → Tạo file thực thi .exe → **Chạy thực thi (terminal)**.

- **Chạy nối tiếp mã nguồn**

Mã nguồn của hệ thống được tổ chức thành file Python, C:

- compiler.py: Chứa code cho lexer, parser, AST, code generator. Đây là file quan trọng nhất. Nó có thể chạy trực tiếp, chấp nhận đối số dòng lệnh: tên file C input và tên file asm output. Khi chạy, nó sẽ sinh file assembly.
- *.c file: Mã nguồn C cần biên dịch (viết theo hạn chế đã nêu).
- Ngoài ra có thể có script shell để nối các bước hoặc ta chạy bằng tay từng bước:
    1. python3 compiler.py input.c output.s
    2. gcc -no-pie output.s -o output
    3. Chạy ./output để thực thi.

  Trước khi chạy, kiểm tra file tồn tại, kiểm tra công cụ cài đặt (gọi gcc --version).

Hướng dẫn sử dụng: Người dùng viết một chương trình C đơn giản (theo cấu trúc đã giới hạn), lưu thành program.c. Sau đó chạy:

```
$ python3 compiler.py program.c program.s
$ gcc -no-pie program.s -o program
$ ./program    # chạy chương trình vừa dịch
```

Nếu mọi thứ thành công, chương trình C sẽ được thực thi. Nếu có lỗi cú pháp hoặc hạn chế, trình biên dịch có thể báo trong quá trình parse (SyntaxError) hoặc assembler có thể báo lỗi nếu mã asm sai. Lúc đó cần xem lại mã nguồn C có vi phạm giới hạn hoặc bug của compiler như đã phân tích.

- **Chạy giao diện đồ họa Code Editor**

File code_editer.py triển khai giao diện đồ họa bằng Tkinter, có các thành phần chính:

- **Menu File: Gồm các lệnh**:
    - New: Xóa nội dung editor, tạo file mới.

- Open: Mở hộp thoại chọn file .c, đọc và hiển thị nội dung.
- Save: Lưu nội dung hiện tại vào file đang mở (nếu đã chọn). Nếu chưa có, gọi Save As.
- Save As: Hộp thoại lưu file với đuôi .c.
- Exit: Thoát chương trình (có hỏi lưu trước khi thoát nếu cần).

- **Khu vực soạn thảo mã (Text):** Cho phép nhập và chỉnh sửa mã C. Kiểm soát có đánh dấu nếu đã sửa và cập nhật tiêu đề cửa sổ.
- **Khu vực console:** Dạng widget Text ở chế độ disabled, hiển thị mã hợp ngữ sinh ra hoặc lỗi (đầu ra của compiler/GCC).
- **Nút chức năng:**
  - **Run Code:** Thực hiện quy trình biên dịch và chạy:
    1. Xóa console, lưu mã C hiện tại vào một file tạm .c.
    2. Gọi subprocess để chạy compiler.py temp.c temp.s. Nếu compiler.py trả về lỗi, hiển thị lỗi trong console và dừng.
    3. Nếu thành công, đọc file .s, hiển thị mã hợp ngữ trong console.
    4. Gọi gcc -no-pie temp.s -o temp để sinh file thực thi. Nếu lỗi GCC, hiển thị lỗi.
    5. Nếu thành công, chắc chắn file thực thi có quyền thực thi, rồi mở một cửa sổ terminal mới (Gnome Terminal) chạy chương trình. Dòng lệnh: bash -c "./temp; echo 'Program finished...'; rm -f temp temp.s temp.c; read".
    6. Cập nhật thanh trạng thái tương ứng (ví dụ Compiler error, GCC error, hay Program running).
  - **Clear Console:** Xóa nội dung khung console.

- **Thanh trạng thái:** Hiển thị trạng thái hiện hành (sẵn sàng, đã lưu, có lỗi, đang chạy, ...).

Cách sử dụng:

1. Chạy python3 code_editer.py. Giao diện GUI hiện ra.
2. Soạn mã C trong ô soạn thảo hoặc mở file .c có sẵn.
3. Lưu file bằng Save hoặc Save As.
4. Nhấn Run Code để biên dịch. Kết quả mã hợp ngữ hiển thị trên console GUI, và chương trình chạy trong terminal riêng biệt.
5. Kiểm tra đầu ra chương trình trong terminal (cửa sổ mới hiện lên). Nhấn Enter để đóng terminal khi chạy xong.
6. Đóng chương trình GUI khi hoàn thành.

Chương 5: Đánh giá, mở rộng và tối ưu hóa

Trong chương này, chúng ta sẽ phân tích những ưu điểm và hạn chế của trình biên dịch vừa xây dựng, thảo luận các hướng mở rộng tính năng để tăng năng lực cho compiler, cũng như đề cập đến các thách thức và kỹ thuật tối ưu hóa có thể áp dụng nếu phát triển xa hơn.

5.1. Đánh giá và hạn chế của hệ thống hiện tại

Ưu điểm & kết quả đạt được:

- **Hoạt động đúng chức năng**: Trình biên dịch đã thực hiện được đầy đủ các bước cần thiết, phân tích, kiểm tra và sinh mã cho hầu hết các câu lệnh và biểu thức C cơ bản đã định nghĩa (khai báo biến, gán, if-else, while, printf, scanf, return). Mã hợp ngữ sinh ra tuân theo kiến trúc x86-64 và có thể được liên kết bằng GCC để tạo file thực thi. Minh chứng là ví dụ đã chạy thành công, cho kết quả đúng. Điều này khẳng định tính đúng đắn cơ bản của cách tiếp cận.

- **Thiết kế module rõ ràng**: Mặc dù code còn giản lược, nhưng kiến trúc chia thành phần (lexer, parser, AST, codegen, assembler, linker) tương đối rõ, thuận tiện cho việc bảo trì hoặc thay thế từng phần. Việc dùng Python giúp mã nguồn ngắn gọn, dễ đọc.
- **Tính trực quan và giáo dục**: Do xử lý một ngôn ngữ rất gần với C thật, nên hệ thống minh họa khá tốt cấu trúc của một trình biên dịch "thật". Điều này có giá trị học thuật, phục vụ cho việc nghiên cứu hoặc giảng dạy (demonstration) về compiler.
- **Độ phức tạp vừa phải**: Số lượng dòng code Python cho compiler front-end khá ít (chưa đến 300 dòng), nhưng bao quát được nhiều khía cạnh (tokenizing, parsing, generating). Điều này đạt được nhờ chấp nhận giới hạn phạm vi. Phạm vi nhỏ cũng giúp dễ kiểm soát lỗi hơn.
- **Giao diện GUI:** Hoạt động ổn định trên Linux. Cho phép nhập mã, lưu/mở file, biên dịch, hiển thị mã assembly và chạy chương trình. Khung console GUI hiển thị thông báo và mã assembly rõ ràng.
- **Hiệu chỉnh kịp thời:** Các lỗi cú pháp/biên dịch (nếu có) được trả về nhanh chóng và hiển thị trong GUI, giúp người dùng sửa mã.

Hạn chế & lỗi hiện tại:

- **Kiểm tra ngữ nghĩa đơn giản:** Hiện chỉ kiểm tra khai báo/biến chưa khai báo và khai báo trùng. Chưa có kiểm tra kiểu chi tiết (ví dụ gán chuỗi cho int hoặc gọi printf với định dạng sai).
- **Giới hạn parser:** Parser chỉ hỗ trợ tối đa một phép toán nhị phân trong một biểu thức. Ví dụ, biểu thức phức a + b - 2 sẽ không được phân tích đầy đủ. Việc kết hợp toán tử chưa hoàn chỉnh.
- **Xử lý gán hạn chế:** Code generator cho phép gán chỉ xử lý khi bên phải là biểu thức nhị phân. Nếu bên phải là hằng số hoặc tên biến đơn thuần, chưa sinh mã, gây thiếu sót.

- **Ngôn ngữ hạn chế:** Chỉ hỗ trợ hàm main, không hỗ trợ hàm con, không có else if, không xử lý pointer ngoài & trong scanf, không hỗ trợ nhiều thư viện ngoài chuẩn C cơ bản.
- **Nền tảng hạn chế:** Giao diện chạy trên Linux với Gnome Terminal. Trên hệ điều hành khác, phần chạy thực thi (dùng gnome-terminal) có thể không hoạt động.
- **Chưa tối ưu mã:** Code generator chưa tối ưu mã (ví dụ không tái sử dụng lệnh, không tối ưu bước tính toán liên tục).

5.2. Các hướng mở rộng tính năng cho compiler

- **Mở rộng tập con ngôn ngữ C**:
    - **Hỗ trợ biểu thức phức tạp hơn**: Nâng cấp parser để hiểu các biểu thức có độ ưu tiên toán tử, có ngoặc tròn trong biểu thức, chuỗi toán tử dài (thay vì chỉ x op y). Điều này đòi hỏi viết bộ phân tích biểu thức (theo phương pháp đệ quy xuống cho các mức ưu tiên hoặc sử dụng giải thuật shunting-yard). Từ đó, AST cho biểu thức sẽ phức tạp (cây nhị phân nhiều tầng) thay vì gói gọn linear. Codegen cũng phải xử lý đệ quy hoặc stack cho biểu thức.
    - **Kiểu dữ liệu khác**: Thêm support cho long, short, unsigned, float, double. Mỗi kiểu sẽ cần xử lý kích thước khác nhau trong .data (hoặc stack). Đặc biệt kiểu dấu phẩy động (float/double) sẽ cần lệnh SSE để tính toán và khác quy ước (dùng XMM regs cho passing).
    - **Mảng và con trỏ**: Hỗ trợ khai báo mảng tĩnh, truy cập mảng (chỉ số), phép toán con trỏ cơ bản (vd: int *p, p = &x, *p = ...). Cái này yêu cầu codegen phức tạp hơn để tính địa chỉ, cung cấp toán tử & (lấy địa chỉ, đã có trong scanf), * (dịch nội dung).
    - **Nhiều hàm, đệ quy**: Cho phép định nghĩa và gọi nhiều hàm khác ngoài main. Lúc đó parser cần quản lý danh sách hàm, mỗi hàm có AST riêng; codegen tạo assembly cho từng hàm, tuân thủ call convention (đẩy tham số lên stack hoặc

thanh ghi tuỳ ABI). Cũng cần hỗ trợ khai báo prototype nếu có, biến cục bộ trong hàm (lưu trên stack frame, chứ không cho hết vào .data).

- **Tăng cường Parser**:
    - Viết parser chắc chắn hơn, có khả năng báo lỗi cụ thể hơn. Có thể xây dựng bảng LR, LALR hoặc sử dụng công cụ (yacc, antlr) để parse chính xác ngữ pháp C. Tuy nhiên, tự viết cũng được với đệ quy xuống nếu cẩn thận sắp xếp thứ tự xử lý.
    - Thêm phân tích ngữ nghĩa: duy trì *bảng ký hiệu (symbol table)* trong quá trình parse. Mỗi khi khai báo biến thì thêm vào bảng (kèm kiểu, kích thước), khi dùng biến thì kiểm tra có trong bảng chưa (nếu chưa, báo lỗi). Quản lý scope: biến trong mỗi hàm, mỗi block {} lồng nhau. Hiện tại, do ta chỉ có global main và cho tất cả biến vào .data (coi như global), việc này đơn giản. Nếu mở rộng, cần hỗ trợ biến cục bộ (phải trừ rsp cấp phát stack, và symbol table để offset).
    - Kiểm tra kiểu (type-checking): hiện int và char chúng ta chẳng phân biệt trong codegen. Nếu có thêm kiểu, cần logic kiểm tra khi gán (loại trừ gán int* cho int chẳng hạn), hoặc khi gọi hàm, v.v.
    - Hỗ trợ *biểu thức quan hệ và logic*: Nếu mở rộng, trong if nên cho phép &&, ||, !. AST cho logic thường là cây riêng (có short-circuit). Codegen cho logic cũng phức tạp hơn (có thể chuyển về các jump).
    - Hỗ trợ *toán tử một ngôi*: như âm (-x), tăng giảm trước/sau (++i, i--), lấy địa chỉ &, nội dung * (dereference). Lexer ta có token ++ -- nhưng parser không dùng. Mở rộng parser để hiểu ++i; thành i = i + 1; về codegen.
- **Cải thiện Code generation**:
    - **Biến cục bộ trên stack**: Hiện ta đặt tất cả biến trong .data (static), dẫn đến chương trình không có biến cục bộ thật sự. Nên cải tiến: khi parse một khai báo trong hàm, thay vì cho vào prog.decls (để .data), ta chỉ lưu trong symbol table và khi codegen, cấp phát trên stack. Cách làm: ví dụ, gặp int x = 5; trong main,

ta sẽ tại codegen thêm sub rsp, 8 (giảm stack pointer 8 byte) để dành chỗ cho x, và lưu giá trị khởi tạo 5 vào [rbp - 8] (offset âm từ rbp). Quản lý offset cho mỗi biến. Cách này phức tạp hơn chút nhưng cho đúng mô hình local. Đối với project nhỏ, có thể vẫn dùng .data (dễ nhưng phi thực tế).

- **Quy hoạch thanh ghi (Register allocation)**: Code hiện tại rất lệ thuộc vào RAX (lấy A vào RAX, tính, rồi lại lưu). Mọi thứ trung chuyển qua RAX và RCX (cho chia). Điều này đơn giản nhưng không tận dụng hết 16 thanh ghi của x86-64. Mở rộng, ta có thể giữ các giá trị trong thanh ghi nếu dùng sau, giảm truy cập bộ nhớ. Ví dụ: a = b + c; d = a * 2; hiện code sẽ: load b, add c, store a; rồi load a, imul 2, store d. Thay vào đó, có thể giữ kết quả b+c trong RAX, rồi dùng luôn RAX * 2 cho d, giảm một lần load store. Kỹ thuật này đòi hỏi phân tích *liveness* và gán thanh ghi thích hợp.
- **Biểu diễn trung gian (IR)**: Đối với một compiler lớn, thường người ta sinh ra code trung gian (như ba địa chỉ) rồi tối ưu rồi mới sinh assembly. Ta có thể thiết kế IR (ví dụ danh sách các tuple kiểu (op, arg1, arg2, dest)) cho chương trình, rồi viết một phần khác để chuyển IR thành ASM. IR giúp tách biệt logic ngôn ngữ nguồn với chi tiết máy, và dễ áp dụng tối ưu (như gộp hằng, loại code thừa). Với quy mô nhỏ, IR không cấp thiết, nhưng nếu mở rộng nhiều tính năng, IR sẽ hữu ích để quản lý phức tạp.
- **Hỗ trợ hàm, stack frame**: Như trên đề cập, codegen thêm prologue/epilogue cho mỗi hàm, xử lý lưu và phục hồi callee-save registers, chuyển tham số từ thanh ghi vào biến cục bộ hoặc thanh ghi, trả về giá trị trong RAX. Quản lý gọi hàm (call) từ code người dùng (hiện chỉ call extern).
- **Hỗ trợ gọi hàm thư viện khác**: Hiện mới có printf, scanf. Có thể mở rộng cho puts, gets, hay exit, v.v. Chỉ cần xử lý parse, codegen call tương tự.

- **Tối ưu hóa**:

- **Loại bỏ mã chết**: Phân tích control-flow để xem những đoạn code nào không bao giờ được thực thi (ví dụ: sau một return vẫn sinh code, nhưng thực ra không cần vì hàm đã kết thúc). Loại những lệnh đó khỏi codegen.
- **Tối ưu phép tính**: Nếu parser/AST nhận biết được hằng, có thể thực hiện tính toán ngay trong compile time (constant folding). Ví dụ int x = 2+3; AST có thể gán x=5 luôn, codegen khỏi sinh add.
- **Tối ưu chuyển điều khiển**: Tránh những jump không cần thiết, ví dụ if then đơn giản không else thì không cần nhãn end; ghép nhiều so sánh liên tiếp v.v.
- **Pipeline tối ưu**: reorder lệnh để tránh phụ thuộc làm chậm CPU pipeline (khá advanced, thường dựa vào phân tích CPU, có lẽ quá mức cho project này).

- **Khả chuyển (portability)**:
  - Hỗ trợ output sang hệ điều hành khác (cách link trên Windows khác, có thể dùng link.exe hoặc gcc -mwindows).
  - Hỗ trợ cross-compile: chạy compiler trên máy này nhưng target máy khác (liên quan đến assembler cross).

- **Giao diện và tiện ích**:
    - **Giao diện:** Hỗ trợ đa nền tảng (ví dụ sử dụng terminal tương thích trên Windows/Mac), highlight cú pháp, bắt lỗi (syntax highlighting, underline lỗi).
    - **Tích hợp công cụ:** Có thể tích hợp với Git hoặc hệ thống quản lý dự án để dễ phát triển.

Rõ ràng, có rất nhiều việc để biến một trình biên dịch đơn giản thành một công cụ mạnh mẽ. Các bước mở rộng nên làm dần dần, kiểm thử kỹ lưỡng, vì thêm tính năng có thể phát sinh lỗi tương tác với phần có sẵn. Ví dụ thêm kiểu mới phải rà soát lại codegen cho toán tử, vv. Trong quá trình mở rộng, cũng học hỏi từ các compiler nguồn mở (như TinyCC, lcc, v.v.) để xem họ tổ chức parser và codegen ra sao.

5.3. Thách thức và kỹ thuật tối ưu hóa

Viết trình biên dịch là một nỗ lực phức tạp. Ngay cả với phiên bản nhỏ này, ta đã thấy một số thách thức (như parse grammar, quản lý ngữ cảnh). Khi mở rộng lên ngôn ngữ đầy đủ và tối ưu, các thách thức tăng đáng kể:

- **Phức tạp của ngôn ngữ nguồn**: Ngôn ngữ C đầy đủ có những phần khó như: tiền xử lý (macro, include), phân tích ngữ cảnh phức tạp (biểu thức phức, cast kiểu, kiểu trừu tượng, ...). Việc xây dựng parser đúng cho toàn bộ C là không hề đơn giản (C có grammar gần LALR(1) nhưng cũng có điểm tricky như *> trong generics C++ càng khó hơn). Do đó, cần áp dụng công cụ parser generator hoặc code cẩn thận.
- **Tối ưu hóa đa dạng**: Có rất nhiều kỹ thuật tối ưu:
    - **Data-flow analysis** (phân tích dòng dữ liệu) để biết biến nào sống (live) tại điểm nào => phục vụ gán thanh ghi, loại biến không dùng.
    - **Control-flow analysis**: xây dựng biểu đồ luồng điều khiển (control flow graph) của chương trình, tìm các khối cơ bản (basic block), tối ưu trong mỗi block và giữa các block.

    - **Global optimization**: như phát hiện biểu thức chung (common subexpression elimination), inline hàm, unroll loop, ... Mỗi kỹ thuật có thuật toán phức tạp (thường dạy ở môn Compiler nâng cao).
    - **Máy ảo hóa**: khi tối ưu ở level cao, có thể compile sang một IR phức tạp hơn (LLVM IR chẳng hạn) để thừa hưởng các tối ưu sẵn có.
- **Quản lý tài nguyên phần cứng**:
    - Thanh ghi CPU có hạn (x86-64 có 16 general regs). Với biểu thức phức tạp hay nhiều biến, cần dùng thuật toán gán thanh ghi (register allocation) hiệu quả, nếu không sẽ phải spill ra stack (ghi tạm ra stack khi thiếu reg) làm chậm chương trình. Thuật toán cổ điển cho việc này là dùng đồ thị màu hóa (graph coloring) để phân phối biến sống đồng thời vào thanh ghi khác nhau.
    - Quản lý pipeline, cache, memory access pattern để code chạy nhanh nhất cũng là lĩnh vực tối ưu (nhưng phần này thường do lập trình viên tối ưu thuật toán, compiler hỗ trợ ở mức sắp xếp lệnh, align dữ liệu).
- **Đảm bảo tính đúng đắn khi tối ưu**: Tối ưu hóa không được làm thay đổi nghĩa chương trình. Nhiều tối ưu phức tạp có thể gây lỗi nếu có trường hợp biên (corner case). Ví dụ tối ưu biến không dùng nhưng nếu biến đó được đọc qua pointer, compiler có thể hiểu sai là biến không xài, xoá mất (sai semantics). Do vậy, compiler phải thận trọng và thường tuân thủ "as-if rule" (chương trình tối ưu chạy *y hệt* kết quả như không tối ưu).
- **Khả năng tương thích và chuẩn**: Với ngôn ngữ C, tuân thủ đúng chuẩn (C89, C99, C11,...) là thách thức. Có nhiều chi tiết: thứ tự đánh giá operand không xác định, integer promotion, v.v. Compiler xịn phải xử lý đúng hết. Hơn nữa, xây dựng trình biên dịch C phải cân nhắc tương thích với hệ thống (ví dụ cho phép inline assembly, tùy chỉnh align memory, attribute như packed struct, v.v.). Phạm vi project nhỏ thường tránh những thứ này, nhưng nếu nâng tầm thì phải giải.
- **Thử nghiệm và kiểm chứng**: Compiler cần được thử nghiệm trên hàng loạt chương trình để đảm bảo không sinh mã sai (một lỗi compiler có thể rất khó dò nếu nó sinh mã máy sai, debug phức tạp hơn debug code thông thường). Viết bộ test, thậm chí formal

verification (kiểm chứng hình thức) với một số thành phần quan trọng (như type-checker) cũng có thể cần nếu làm compiler cho hệ thống an toàn.

Một vài kỹ thuật và hướng giải quyết:

- Sử dụng công cụ & thư viện có sẵn: ví dụ, tái sử dụng front-end Clang để parse C, hoặc sử dụng LLVM làm backend, như vậy chúng ta chỉ viết phần "dán keo" (glue) nhưng hưởng lợi từ hàng thập kỷ phát triển. Tuy nhiên, làm vậy thì mục đích học tập giảm vì ta dùng sẵn nhiều.
- Thiết kế ngôn ngữ trung gian: lối tiếp cận của LLVM là tạo ra một IR rất mạnh và tối ưu trên IR đó, sau mới sinh mã máy. Trong project cá nhân, ta có thể thiết kế IR đơn giản (3-address code, stack machine code, etc.) và viết các passes tối ưu IR rồi generate. IR nên đủ trừu tượng để dễ tối ưu (VD: có infinite virtual registers rồi sau đó allocate to real registers).
- Từng bước phát triển, tối ưu cục bộ trước: Tập trung làm chạy đúng trước (O0), sau đó thêm dần tối ưu mức O1, O2... Viết code rõ ràng, tách thành các hàm nhỏ (module) để sau này thay thế/ cải tiến dễ.
- Đọc các thuật toán kinh điển: Sách "Compilers: Principles, Techniques, and Tools" (Aho, Sethi, Ullman) – còn gọi là "Dragon Book" – là nguồn kinh điển cho nhiều thuật toán từ lexer, parser LR, đến tối ưu DAG, register allocation, etc. Ngoài ra có sách hiện đại như "Engineering a Compiler" hoặc "Modern Compiler Implementation in C/Java". Áp dụng những thuật toán này sẽ nâng chất lượng compiler.
- Sử dụng Profiling và Benchmark: Để biết chỗ nào mã sinh ra còn chưa tốt, ta có thể compile một số chương trình, dùng profiler khi chạy để xem nóng ở đâu, hoặc so sánh mã máy với mã của GCC ở mức -O0, -O2,... để học hỏi.

Trong phạm vi đồ án này, những thách thức tối ưu hóa sâu chưa cần giải quyết triệt để, nhưng nhìn ra trước các vấn đề sẽ giúp định hướng khi mở rộng, phát triển trình biên dịch.

Chương 6: Kết luận

6.1. Kết quả đạt được:

Chúng em đã xây dựng thành công một trình biên dịch C cơ bản bằng Python, có khả năng dịch một tập con nhỏ của C sang mã máy và thực thi. Hệ thống gồm các phần: bộ phân tích từ vựng sử dụng regex, bộ phân tích cú pháp dạng đệ quy xuống, cấu trúc AST đơn giản, và bộ sinh mã hợp ngữ cho kiến trúc x86-64. Mặc dù đơn giản, trình biên dịch hoạt động đúng trên các chương trình mẫu thử nghiệm, cho thấy sự phối hợp chính xác giữa các giai đoạn biên dịch và việc tuân thủ quy ước hệ thống (như gọi hàm thư viện). Kết quả thực thi chương trình đã biên dịch khớp với mong đợi, chứng minh tính đúng đắn của compiler ở mức độ tính năng cho phép.

Qua quá trình thực hiện, đồ án đã giúp đào sâu hiểu biết về cách một trình biên dịch hoạt độn. Từ việc tách chuỗi mã nguồn thành token, đến việc xây dựng cây cấu trúc và cuối cùng là phát sinh các lệnh máy cụ thể – tất cả đều được hiện thực hoá bằng lập trình, thay vì chỉ qua lý thuyết. Việc lựa chọn Python làm ngôn ngữ cài đặt tỏ ra hiệu quả cho mục đích trình bày thuật toán rõ ràng và thử nghiệm nhanh. Bên cạnh đó, tích hợp các công cụ như NASM và GCC cho phần hậu xử lý cũng minh họa chu trình biên dịch đầy đủ trong môi trường thực tế.

6.2. Hướng phát triển

Mặc dù đạt được chức năng cơ bản, chương trình vẫn còn ở quy mô thử nghiệm. Trong tương lai, có thể phát triển theo các định hướng sau:

- **Mở rộng ngôn ngữ nguồn**: Tiếp tục hoàn thiện hỗ trợ ngôn ngữ C: thêm vòng lặp, mảng, con trỏ, hàm, v.v. Điều này sẽ đưa trình biên dịch gần hơn với một compiler C hoàn chỉnh, đồng thời tăng độ phức tạp và cần thiết kế cấu trúc dữ liệu, giải thuật mạnh mẽ hơn (ví dụ: bảng ký hiệu, quản lý vùng nhớ stack).
- **Cải thiện tính đúng đắn và ổn định**: Nâng cấp bộ phân tích cú pháp để xử lý được nhiều tình huống hợp lệ hơn và đưa ra thông báo lỗi tốt hơn khi gặp lỗi. Bổ sung kiểm tra ngữ nghĩa (khai báo, kiểu) để phát hiện lỗi sớm. Việc này nâng cao trải nghiệm sử dụng và độ tin cậy.

- **Tối ưu hóa mã sinh**: Triển khai một số tối ưu cơ bản như đã bàn (loại mã thừa, gộp hằng, dùng thanh ghi, ...). Mục tiêu là để chương trình biên dịch ra mã chạy hiệu quả hơn, học hỏi các kỹ thuật tối ưu của compiler thực sự.
- **Hỗ trợ đa kiến trúc**: Thử nghiệm sinh mã cho kiến trúc khác (ví dụ x86-32bit, ARM) trong cùng framework, tăng tính linh hoạt và hiểu biết về khác biệt kiến trúc.
- **Xây dựng trình biên dịch cho ngôn ngữ khác**: Từ kinh nghiệm với C subset, có thể thử xây dựng trình biên dịch cho một ngôn ngữ tự thiết kế hoặc ngôn ngữ bậc cao khác. Python codebase có thể tái sử dụng một phần (ví dụ module lexer có thể chỉnh lại regex cho ngôn ngữ khác).
- **Học sâu hơn về compiler**: Sử dụng dự án này như bàn đạp để nghiên cứu sâu thuật toán phân tích cú pháp (LR parser), tối ưu toàn cục, hoặc tham gia đóng góp cho các compiler open-source để cọ xát với codebase lớn.

6.3. Tài liệu tham khảo:

- Aho, A. V., Sethi, R., & Ullman, J. D. (1986). Compilers: Principles, Techniques, and Tools. Addison-Wesley.
- Các tài liệu trực tuyến về Recursive Descent Parsing và Lexical Analysis.
- Writing a C compiler in 500 lines, August 30, 2023 https://vgel.me/posts/c500/
- Richard Bornat, Understanding and Writing Compilers: A Do It Yourself Guide