**Giới thiệu**

**Tại sao phải xây dựng trình biên dịch, vai trò của nó**

Trình biên dịch là công cụ không thể thiếu trong lĩnh vực công nghệ thông tin, giúp chuyển đổi mã nguồn từ ngôn ngữ lập trình cấp cao sang mã máy hoặc mã trung gian để máy tính có thể hiểu và thực thi. Việc xây dựng trình biên dịch không chỉ nhằm mục đích sử dụng các ngôn ngữ lập trình một cách hiệu quả mà còn mở rộng khả năng tối ưu hóa mã nguồn, tăng cường hiệu suất và giảm thiểu lỗi trong quá trình phát triển phần mềm.

Trình biên dịch đóng vai trò như một cầu nối giữa lập trình viên và phần cứng máy tính. Nó giúp:

* Tự động hóa việc chuyển đổi mã nguồn, giảm thiểu can thiệp thủ công.
* Tối ưu hóa tài nguyên phần cứng và hiệu năng chương trình.
* Tạo ra phần mềm có tính di động cao, dễ dàng chạy trên nhiều nền tảng khác nhau.

**Tổng quan về xây dựng trình biên dịch ngôn ngữ BASIC sang LLVM-IR**

Ngôn ngữ BASIC được biết đến với cú pháp đơn giản, dễ học, phù hợp để xây dựng các ứng dụng cơ bản. Kết hợp với LLVM, một hạ tầng biên dịch mạnh mẽ, việc xây dựng trình biên dịch từ BASIC sang LLVM-IR mang lại khả năng tối ưu hóa sâu sắc, tạo điều kiện cho chương trình BASIC hoạt động hiệu quả trên nhiều nền tảng khác nhau.

Quá trình này bao gồm:

* Phân tích mã nguồn BASIC để tạo cây cú pháp trừu tượng (AST).
* Chuyển đổi AST sang dạng biểu diễn trung gian LLVM-IR.
* Tối ưu hóa mã trung gian và sinh mã máy để thực thi hiệu quả.

Việc tích hợp BASIC và LLVM không chỉ giúp tận dụng sự đơn giản của BASIC mà còn mang lại sức mạnh xử lý và tối ưu hóa của LLVM.

**Mục đích của khóa học và các giai đoạn chính**

Khóa học tập trung vào việc cung cấp kiến thức nền tảng và kỹ năng thực tiễn để xây dựng trình biên dịch. Các mục tiêu chính bao gồm:

* Hiểu các khái niệm cơ bản về phân tích từ vựng, cú pháp và ngữ nghĩa.
* Nắm bắt quy trình chuyển đổi mã nguồn sang biểu diễn trung gian (IR).
* Thực hiện tối ưu hóa và sinh mã đích, đảm bảo hiệu quả và tính chính xác.

Các giai đoạn chính của khóa học:

1. **Phân tích mã nguồn:** Tách mã nguồn thành các thành phần cơ bản và kiểm tra tính hợp lệ.
2. **Xây dựng cây cú pháp trừu tượng (AST):** Đại diện chương trình bằng một cấu trúc cây logic.
3. **Sinh mã trung gian (LLVM-IR):** Chuyển đổi AST thành mã trung gian phù hợp với LLVM.
4. **Tối ưu hóa mã:** Cải thiện hiệu suất của chương trình bằng cách loại bỏ các phần dư thừa hoặc không hiệu quả.
5. **Sinh mã máy:** Tạo mã thực thi phù hợp với nền tảng đích.

Khóa học không chỉ trang bị kiến thức lý thuyết mà còn thực hành các bước để xây dựng một trình biên dịch hoàn chỉnh, giúp học viên hiểu sâu hơn về cấu trúc và nguyên lý hoạt động của ngôn ngữ lập trình và hệ thống máy tính.

**1. Phần lý thuyết**

**1.1 Giới thiệu ngôn ngữ Basic và các đặc điểm của nó**

**Quá trình phát triển của ngôn ngữ BASIC**

Ngôn ngữ BASIC (Beginner's All-purpose Symbolic Instruction Code) được phát triển vào năm 1964 bởi John G. Kemeny và Thomas E. Kurtz tại Đại học Dartmouth với mục tiêu cung cấp một công cụ lập trình đơn giản và dễ học cho người dùng không chuyên về khoa học máy tính. BASIC đã trải qua quá trình phát triển qua nhiều giai đoạn:

1. **Giai đoạn khởi đầu (1964-1970):**

BASIC ra đời như một ngôn ngữ phục vụ giảng dạy và vận hành trên hệ thống chia sẻ thời gian Dartmouth Time-Sharing System (DTSS). Đây là bước tiến lớn giúp đưa lập trình tới gần hơn với các lĩnh vực ngoài khoa học máy tính.

1. **Giai đoạn phổ biến trên máy tính cá nhân (1970-1980):**

BASIC trở thành ngôn ngữ lập trình phổ biến nhất trên các máy tính cá nhân như Altair 8800, Apple II, Commodore 64, và các hệ thống tương tự. Microsoft BASIC, do Bill Gates và Paul Allen phát triển, đóng vai trò then chốt trong việc đưa BASIC lên các hệ điều hành phổ biến như CP/M và MS-DOS.

1. **Giai đoạn cải tiến và mở rộng (1980-1990):**

Trong giai đoạn này, BASIC được phát triển với nhiều phiên bản mở rộng, hỗ trợ lập trình cấu trúc và giao diện người dùng đồ họa (GUI). Visual Basic, ra mắt năm 1991 bởi Microsoft, là một trong những thành tựu đáng chú ý, tích hợp hỗ trợ lập trình hướng đối tượng và xây dựng ứng dụng đồ họa.

1. **Giai đoạn suy thoái và vai trò lịch sử (1990 đến nay):**

Với sự phát triển của các ngôn ngữ lập trình hiện đại như Python, Java, và C++, BASIC dần mất đi sự phổ biến. Tuy nhiên, nó vẫn được ghi nhận là một trong những ngôn ngữ khởi nguồn, đóng góp quan trọng vào việc phổ cập lập trình trên toàn cầu.

**Các phiên bản đáng chú ý của BASIC**

1. **Dartmouth BASIC (1964):** Phiên bản nguyên bản, tập trung vào tính đơn giản, vận hành trên hệ thống DTSS.
2. **Microsoft BASIC (1975):** Một trong những phiên bản phổ biến nhất, được phát triển để chạy trên các hệ thống CP/M, MS-DOS và trở thành chuẩn mực trên nhiều máy tính cá nhân thời kỳ đầu.
3. **GW-BASIC (1983):** Một biến thể của Microsoft BASIC, hỗ trợ nhiều tính năng mới và tích hợp môi trường phát triển cơ bản cho máy tính cá nhân.
4. **QBASIC (1991):** Phiên bản cải tiến với giao diện thân thiện hơn, hỗ trợ lập trình cấu trúc, phổ biến trên MS-DOS.
5. **Visual Basic (1991):** Một trong những phiên bản thành công nhất, được tối ưu hóa để xây dựng ứng dụng đồ họa, hỗ trợ lập trình hướng đối tượng và tích hợp phát triển giao diện người dùng đồ họa.
6. **FreeBASIC (2004):** Một phiên bản hiện đại mã nguồn mở, tương thích với QBASIC và hỗ trợ các tính năng của lập trình cấu trúc và lập trình hướng đối tượng.

**Ý nghĩa và vai trò của BASIC**

Ngôn ngữ BASIC không chỉ là công cụ học tập hiệu quả mà còn đóng vai trò quan trọng trong việc phổ cập lập trình cho mọi tầng lớp người dùng, từ học sinh, sinh viên đến các nhà nghiên cứu. Sự phát triển qua nhiều phiên bản đã minh chứng cho tính ứng dụng linh hoạt và khả năng thích nghi với các yêu cầu công nghệ khác nhau trong lịch sử.

Hiện nay, mặc dù không còn phổ biến như trước, nhưng việc nghiên cứu và áp dụng BASIC, chẳng hạn trong biên dịch sang LLVM-IR, vẫn là cách để bảo tồn di sản và khám phá những giá trị tiềm năng của ngôn ngữ này trong bối cảnh công nghệ hiện đại.

**Các đặc điểm nổi bật của BASIC:**

1. **Cú pháp đơn giản:**

BASIC được thiết kế với cú pháp gần gũi và dễ hiểu, phù hợp cho những người mới làm quen với lập trình. Ví dụ, các câu lệnh thường sử dụng từ khóa trực quan như PRINT, INPUT, IF...THEN.

1. **Di động và phổ biến:**

BASIC có thể được triển khai trên nhiều nền tảng khác nhau, từ máy tính cá nhân đến các hệ thống nhúng. Các phiên bản khác nhau của BASIC, như QBASIC, Visual Basic, đã được phát triển để phù hợp với các nhu cầu ứng dụng khác nhau.

1. **Hỗ trợ lập trình tuyến tính:**

BASIC ban đầu được thiết kế để hỗ trợ lập trình tuyến tính, giúp người dùng dễ dàng hiểu được luồng chương trình. Các phiên bản sau này bổ sung hỗ trợ cho lập trình cấu trúc và lập trình hướng đối tượng.

1. **Thân thiện với người dùng:**

Ngôn ngữ này nhấn mạnh vào khả năng dễ học và thân thiện với người dùng, cung cấp nhiều công cụ hỗ trợ cho việc kiểm tra lỗi và gỡ rối.

1. **Ứng dụng rộng rãi:**

BASIC được sử dụng rộng rãi trong giáo dục và các ứng dụng đơn giản. Mặc dù không còn phổ biến như trước, nhưng nó vẫn là nền tảng cho các ngôn ngữ lập trình hiện đại như Visual Basic.

Bình luận bắt đầu bằng dấu nháy đơn ' và tiếp tục cho đến cuối dòng.

Mã định danh và từ khóa không phân biệt chữ hoa chữ thường.

Tên của mỗi biến và mỗi hàm chỉ rõ kiểu của nó (% — số nguyên, & —số nguyên dài, ! — thực độ chính xác đơn, # — thực độ chính xác kép độ chính xác, $ là một chuỗi).

Bên trong hàm, một biến được khai báo ngầm định với tên giống như tên của hàm đó.

hàm - giá trị của nó là giá trị trả về của hàm.

Cả lập chỉ mục mảng và gọi hàm đều được viết bằng ký hiệu vòng.

dấu ngoặc vuông - không thể phân biệt được dấu ngoặc vuông này với dấu ngoặc vuông kia ở giai đoạn phân tích cú pháp,

do đó, trên cây chúng không nên khác nhau.

Vòng lặp while có thể được viết theo năm cách:

A close up of words

Description automatically generated

Hai dạng đầu tiên là các chu kỳ có điều kiện tiên quyết (tích cực và tiêu cực),

hai cái còn lại có điều kiện hậu kỳ, và cái cuối cùng là cái thứ năm là một vòng lặp vô hạn.

Vòng lặp For có thể bị ngắt bởi các câu lệnh

A close up of words

Description automatically generated

Biểu mẫu đầu tiên ngắt vòng lặp hiện tại, biểu mẫu thứ hai (có biến) cho phép bạn ngắt nhiều vòng lặp lồng nhau cùng một lúc.

Chu trình Do/Loop có thể bị ngắt bởi một trong hai câu lệnh sau (chúng là từ đồng nghĩa):

A close up of words

Description automatically generated

Thủ tục và hàm được kết thúc tương ứng bằng các câu lệnh.

A close-up of a white background

Description automatically generated

Bạn có thể định nghĩa biến toàn cục:

A black text on a white background

Description automatically generated

Bạn có thể định nghĩa mảng đa chiều:



**1.2 Cơ bản các bước xây dựng trình biên dịch**

Việc xây dựng một trình biên dịch là một quy trình tuần tự, bao gồm các bước cơ bản, mỗi bước đóng vai trò quan trọng trong việc chuyển đổi mã nguồn từ ngôn ngữ cấp cao sang mã đích. Giai đoạn đầu tiên trong quy trình này là **phân tích từ vựng**, nơi mã nguồn được xử lý để nhận diện các thành phần cơ bản của ngôn ngữ.

**1.2.1. Phân tích từ vựng**

**Phân tích từ vựng (Lexical Analysis)** là giai đoạn đầu tiên của trình biên dịch, có nhiệm vụ đọc mã nguồn đầu vào và chuyển đổi nó thành một chuỗi các **token** – các đơn vị từ vựng cơ bản, được xác định dựa trên cú pháp của ngôn ngữ lập trình.

**Mục tiêu của phân tích từ vựng:**

1. **Nhận diện từ vựng:** Phân tách mã nguồn thành các thành phần cơ bản như từ khóa, toán tử, dấu chấm câu, tên biến, và giá trị số.
2. **Loại bỏ các phần không cần thiết:** Bỏ qua các ký tự không ảnh hưởng đến chương trình như khoảng trắng, dòng trống và chú thích.
3. **Cung cấp đầu ra:** Tạo danh sách các token kèm thông tin về loại token, giá trị và vị trí trong mã nguồn để cung cấp cho giai đoạn tiếp theo (phân tích cú pháp).

**Các giai đoạn của phân tích từ vựng**

1. **Đọc mã nguồn:**  
   Mã nguồn được xử lý dưới dạng một chuỗi ký tự. Trình biên dịch sẽ đọc tuần tự từng ký tự để nhận diện các mẫu từ vựng.
2. **Nhận diện mẫu từ vựng (Lexeme):**  
   Dựa trên các quy tắc và mẫu được định nghĩa trong đặc tả từ vựng của ngôn ngữ (thường biểu diễn bằng biểu thức chính quy), các chuỗi ký tự được nhận diện là các lexeme. Ví dụ: int, if, x, 123.
3. **Chuyển đổi lexeme thành token:**  
   Mỗi lexeme được ánh xạ tới một loại token. Token là một cấu trúc dữ liệu gồm ba thành phần chính:
   * **Loại token** (Token Type): Ví dụ, KEYWORD, IDENTIFIER, NUMBER.
   * **Giá trị token** (Token Value): Giá trị của lexeme, ví dụ, x hoặc 123.
   * **Vị trí token** (Position): Vị trí lexeme trong mã nguồn, giúp định vị lỗi nếu cần.
4. **Xử lý lỗi từ vựng:**  
   Nếu gặp lexeme không phù hợp với quy tắc ngôn ngữ, bộ phân tích từ vựng sẽ báo lỗi và chỉ ra vị trí lỗi trong mã nguồn để người dùng sửa chữa.
5. **Cấp phát token:**  
   Chuỗi token cuối cùng được gửi đến bộ phân tích cú pháp (Syntax Analyzer) để tiếp tục xử lý.

**1.2.2. Phân tích cú pháp (xây dựng cây phân tích cú pháp)**

**Phân tích cú pháp (Syntax Analysis)** là giai đoạn tiếp theo trong quy trình xây dựng trình biên dịch. Giai đoạn này kiểm tra cấu trúc cú pháp của mã nguồn đầu vào dựa trên các quy tắc ngữ pháp chính thức của ngôn ngữ lập trình. Kết quả đầu ra là một **cây phân tích cú pháp (Parse Tree)**, biểu diễn cấu trúc lô-gic và quan hệ giữa các thành phần của mã nguồn.

**Mục tiêu của phân tích cú pháp**

1. **Kiểm tra tính hợp lệ cú pháp:** Đảm bảo mã nguồn tuân thủ các quy tắc cú pháp của ngôn ngữ.
2. **Xây dựng cây phân tích cú pháp:** Tạo một cấu trúc cây biểu diễn quan hệ giữa các token trong mã nguồn.
3. **Cung cấp đầu ra cho giai đoạn phân tích ngữ nghĩa:** Đầu ra là cây phân tích cú pháp, được sử dụng để kiểm tra ý nghĩa và các ràng buộc ngữ nghĩa của chương trình.

**Các giai đoạn của phân tích cú pháp**

1. **Nhập chuỗi token:**  
   Chuỗi token từ giai đoạn phân tích từ vựng được gửi đến bộ phân tích cú pháp để xử lý.
2. **Kiểm tra quy tắc ngữ pháp:**
   * Sử dụng các quy tắc ngữ pháp phi ngữ cảnh (Context-Free Grammar - CFG) để kiểm tra xem chuỗi token có tuân thủ cú pháp của ngôn ngữ không.
   * Các quy tắc ngữ pháp thường được biểu diễn bằng văn phạm Backus-Naur Form (BNF) hoặc Extended BNF (EBNF).
3. **Xử lý cú pháp và xây dựng cây phân tích cú pháp:**
   * Dựa trên các quy tắc ngữ pháp, bộ phân tích cú pháp sẽ xây dựng một cây phân tích cú pháp (Parse Tree).
   * Mỗi **node** trong cây đại diện cho một cấu trúc cú pháp, với các **node lá** biểu diễn các token cụ thể (như từ khóa, toán tử, hoặc tên biến).
4. **Phát hiện và xử lý lỗi cú pháp:**
   * Khi phát hiện mã nguồn không tuân thủ cú pháp, bộ phân tích cú pháp sẽ báo lỗi, chỉ ra vị trí và loại lỗi.
   * Một số trình biên dịch sử dụng **cơ chế phục hồi lỗi (Error Recovery)** để tiếp tục phân tích thay vì dừng hoàn toàn.
5. **Tối ưu hóa cây phân tích cú pháp (tùy chọn):**
   * Tối giản cây phân tích cú pháp bằng cách loại bỏ các node không cần thiết hoặc hợp nhất các node trung gian, tạo ra **Cây cú pháp trừu tượng (Abstract Syntax Tree - AST)**.
   * AST thường được sử dụng làm đầu vào cho giai đoạn phân tích ngữ nghĩa.

**1.2.3. Phân tích ngữ nghĩa (Xây dựng cây cú pháp trừu tượng)**

**Phân tích ngữ nghĩa (Semantic Analysis)** là giai đoạn tiếp theo sau phân tích cú pháp. Giai đoạn này đảm bảo rằng mã nguồn không chỉ đúng về cú pháp mà còn hợp lý về ý nghĩa theo quy tắc của ngôn ngữ lập trình. Kết quả của giai đoạn này là một **cây cú pháp trừu tượng (Abstract Syntax Tree - AST)**, được bổ sung các thông tin ngữ nghĩa.

**Mục tiêu của phân tích ngữ nghĩa**

1. **Xác minh tính hợp lệ ngữ nghĩa của mã nguồn:** Đảm bảo rằng các câu lệnh, biểu thức và thao tác đều có ý nghĩa logic và đúng quy tắc của ngôn ngữ.
2. **Xây dựng cây cú pháp trừu tượng (AST):** AST là phiên bản tối giản và tối ưu hơn của cây phân tích cú pháp, biểu diễn logic của chương trình một cách dễ xử lý hơn.
3. **Chuẩn bị cho các giai đoạn tiếp theo:** Giai đoạn này tạo các đại diện trung gian nội bộ để hỗ trợ dịch mã nguồn sang mã máy hoặc mã mục tiêu.

**Các bước chính của phân tích ngữ nghĩa**

1. **Phân tích các kiểu dữ liệu:**
   * Kiểm tra kiểu dữ liệu của các biểu thức, biến, và giá trị trả về từ hàm.
   * Phát hiện và xử lý các lỗi không tương thích kiểu dữ liệu, ví dụ: cộng chuỗi với số.
2. **Kiểm tra nội dung khai báo và phạm vi:**
   * Đảm bảo rằng tất cả các biến, hằng số, hàm, và thủ tục đều được khai báo trước khi sử dụng.
   * Xác minh rằng các biến được sử dụng đúng phạm vi (scope), tránh xung đột tên hoặc tham chiếu không hợp lệ.
3. **Phân giải tên (Name Resolution):**
   * Liên kết các tên (tên biến, hàm, thủ tục) với các đối tượng tương ứng trong bảng ký hiệu (Symbol Table).
   * Giải quyết các tham chiếu, đặc biệt trong trường hợp có sự che khuất tên giữa các phạm vi khác nhau.
4. **Phân tích cấu trúc điều khiển:**
   * Kiểm tra tính hợp lệ của các câu lệnh điều kiện (IF, FOR, WHILE) và vòng lặp.
   * Đảm bảo các biểu thức điều kiện trả về giá trị hợp lệ (thường là boolean).
5. **Lời gọi hàm, thủ tục:**
   * Kiểm tra xem các lời gọi hàm có số lượng và kiểu tham số phù hợp với định nghĩa của hàm.
   * Đảm bảo giá trị trả về từ hàm được sử dụng đúng cách.
6. **Kiểm tra các hạn chế ngữ nghĩa của ngôn ngữ:**
   * Xác minh các ràng buộc ngữ nghĩa đặc thù của ngôn ngữ, như cấm thay đổi hằng số, chỉ cho phép kiểu dữ liệu cụ thể trong một ngữ cảnh.
7. **Tạo đại diện nội bộ:**
   * Giai đoạn này chuyển đổi cây cú pháp trừu tượng (AST) thành biểu diễn trung gian (Intermediate Representation - IR).
   * IR là một dạng biểu diễn dễ dàng chuyển đổi sang mã máy hoặc mã mục tiêu.

**Cây cú pháp trừu tượng (AST)**

**AST** là phiên bản rút gọn của cây phân tích cú pháp, chỉ giữ lại các thông tin cần thiết để biểu diễn logic của chương trình. AST loại bỏ các node dư thừa và tập trung vào các thành phần ngữ nghĩa cốt lõi, như:

* Các câu lệnh điều kiện.
* Biểu thức toán học và logic.
* Cấu trúc vòng lặp.

**2.2.4. Xây dựng bộ dịch sang mã Assembly**

Giai đoạn xây dựng bộ dịch sang mã Assembly là bước chuyển đổi từ biểu diễn trung gian (Intermediate Representation - IR) sang mã máy hoặc mã Assembly. Đây là giai đoạn tập trung vào việc tối ưu hóa hiệu suất và đảm bảo mã đầu ra có thể thực thi hiệu quả trên phần cứng hoặc nền tảng mục tiêu.

**Các bước chính trong xây dựng bộ dịch sang mã Assembly**

1. **Lựa chọn cấu trúc mục tiêu:**
   * Xác định kiến trúc phần cứng hoặc môi trường thực thi (CPU, GPU, LLVM, v.v.).
   * Tối ưu hóa mã Assembly cho các đặc điểm kỹ thuật cụ thể của nền tảng (ví dụ: số thanh ghi, tập lệnh, hoặc mô hình bộ nhớ).
2. **Chuyển biểu diễn bên trong sang mã máy (Machine Code):**
   * Sử dụng **biểu diễn trung gian (IR)** từ giai đoạn trước để tạo mã Assembly.
   * Chuyển đổi các cấu trúc điều khiển (IF, FOR, WHILE) và biểu thức thành các lệnh máy cụ thể.
   * Gán các biến và giá trị vào thanh ghi hoặc vị trí trong bộ nhớ.
   * Xử lý các thao tác toán học, logic và truy cập bộ nhớ thành các lệnh Assembly tương ứng.
3. **Tạo dữ liệu bổ sung:**
   * **Bảng ký hiệu (Symbol Table):** Lưu trữ thông tin về vị trí các biến, hàm, và nhãn trong bộ nhớ.
   * **Bảng nhãn (Label Table):** Lưu trữ các điểm nhảy trong mã Assembly để hỗ trợ điều khiển dòng (goto, loop).
   * **Hỗ trợ gỡ lỗi (Debugging Support):** Thêm thông tin giúp theo dõi mã nguồn trong quá trình gỡ lỗi, chẳng hạn ánh xạ từ mã nguồn sang mã Assembly.
4. **Hỗ trợ môi trường thời gian chạy (Runtime Environment):**
   * Cung cấp các thành phần cần thiết để hỗ trợ thực thi mã biên dịch, bao gồm:
     + **Khởi tạo chương trình:** Thiết lập bộ nhớ, stack, heap, và các biến toàn cục.
     + **Hàm thư viện:** Kết nối mã với các thư viện runtime hoặc hệ điều hành, cung cấp các dịch vụ như I/O, quản lý bộ nhớ, hoặc gọi hàm hệ thống.
     + **Xử lý lỗi:** Thêm các cơ chế phát hiện và xử lý ngoại lệ trong mã Assembly.
   * Đảm bảo rằng mã Assembly tương thích với hệ điều hành và trình liên kết (Linker) trong hệ thống mục tiêu.

**2.2.5. Xây dựng Bố cục**

Giai đoạn xây dựng bố cục là quá trình cuối cùng trong việc tạo ra tệp thực thi hoặc thư viện từ mã nguồn. Trong giai đoạn này, tất cả các thành phần của chương trình được tổ chức và liên kết lại thành một sản phẩm cuối cùng có thể chạy trên môi trường mục tiêu. Các công việc chính trong giai đoạn này bao gồm lựa chọn và tải các mô đun, tạo bảng ký hiệu, và tạo tệp thực thi hoặc thư viện.

**Các bước chính trong xây dựng bố cục**

1. **Lựa chọn và tải các mô đun:**
   * **Mô đun (Modules):** Trong giai đoạn này, các mô đun chương trình được chọn và tải vào không gian bộ nhớ. Mỗi mô đun thường đại diện cho một thành phần cụ thể của chương trình (ví dụ: các hàm, thủ tục, thư viện).
   * Quá trình này có thể bao gồm:
     + **Liên kết động (Dynamic Linking):** Các mô đun được tải khi chương trình chạy và có thể thay đổi theo môi trường.
     + **Liên kết tĩnh (Static Linking):** Các mô đun được kết nối trực tiếp vào tệp thực thi trong quá trình biên dịch.
   * Tùy thuộc vào cách thức xây dựng, các mô đun có thể được liên kết ngay khi biên dịch hoặc được liên kết khi thực thi.
2. **Tạo bảng ký hiệu (Symbol Table):**
   * **Bảng ký hiệu** chứa thông tin về tất cả các thành phần trong chương trình, bao gồm biến, hằng số, hàm, thủ tục, và các đối tượng khác.
   * Bảng ký hiệu lưu trữ các thông tin như tên, kiểu dữ liệu, phạm vi, vị trí bộ nhớ của các thành phần này. Đây là nguồn thông tin quan trọng để liên kết các mô đun và thực hiện các thao tác trong chương trình.
   * Trong giai đoạn xây dựng bố cục, bảng ký hiệu được sử dụng để xác định vị trí và cách thức kết nối các thành phần của chương trình.
3. **Tạo tệp hoặc thư viện thực thi:**
   * **Tạo tệp thực thi:** Sau khi tất cả các mô đun được tải và bảng ký hiệu được tạo ra, trình biên dịch sẽ tạo ra tệp thực thi. Tệp này có thể là một tệp **.exe** hoặc **.out** (tuỳ thuộc vào hệ điều hành). Tệp thực thi này có thể được chạy trực tiếp trong môi trường mục tiêu.
   * **Tạo thư viện:** Ngoài tệp thực thi, trong một số trường hợp, một **thư viện động (DLL)** hoặc **thư viện tĩnh** có thể được tạo ra. Thư viện này chứa các mô đun chung mà có thể được các chương trình khác sử dụng. Thư viện giúp tái sử dụng mã nguồn và giảm thiểu sự trùng lặp trong các ứng dụng.

**2.2.6.** **Tổng kết**

Quá trình xây dựng một trình biên dịch từ mã nguồn ngôn ngữ lập trình sang mã máy hoặc mã Assembly là một quá trình phức tạp và yêu cầu sự phối hợp của nhiều giai đoạn khác nhau. Bắt đầu từ phân tích từ vựng, nơi mà mã nguồn được chia thành các đơn vị nhỏ để dễ dàng xử lý, qua việc phân tích cú pháp để xây dựng cấu trúc cây phân tích cú pháp, tiếp theo là phân tích ngữ nghĩa để kiểm tra tính hợp lệ của chương trình về mặt ngữ nghĩa và kiểu dữ liệu. Sau đó, trình biên dịch chuyển đổi mã nguồn thành một biểu diễn trung gian, tối ưu hóa và cuối cùng chuyển sang mã Assembly hoặc mã máy.

Quá trình xây dựng bộ dịch sang mã Assembly giúp tối ưu hóa hiệu suất và tạo ra mã máy tương thích với phần cứng mục tiêu. Cuối cùng, bước xây dựng bố cục tổ chức và liên kết các mô đun, tạo bảng ký hiệu và tạo tệp thực thi hoặc thư viện giúp hoàn thiện sản phẩm biên dịch, sẵn sàng cho quá trình triển khai và thực thi.

Mỗi giai đoạn trong quá trình xây dựng trình biên dịch đóng vai trò quan trọng trong việc đảm bảo tính chính xác, hiệu quả và khả năng tái sử dụng của phần mềm. Từ đó, các chương trình có thể được chuyển đổi và thực thi trên các hệ thống mục tiêu một cách hiệu quả và ổn định.

**2. Triển khai và phát triển**

**2.1 Lựa chọn ngôn ngữ lập trình**

Trong quá trình phát triển trình biên dịch cho ngôn ngữ Basic, việc lựa chọn ngôn ngữ lập trình và công cụ hỗ trợ đóng vai trò quan trọng trong việc đảm bảo hiệu quả và tính khả thi của dự án. Sau khi cân nhắc nhiều yếu tố như tính linh hoạt, cộng đồng hỗ trợ, và khả năng phát triển nhanh chóng, Python đã được chọn làm ngôn ngữ chính để xây dựng trình biên dịch.

Python là một ngôn ngữ lập trình bậc cao với cú pháp rõ ràng, dễ đọc và dễ học, giúp giảm thiểu thời gian phát triển và tăng cường khả năng bảo trì mã nguồn. Python rất phù hợp để triển khai các ứng dụng liên quan đến phân tích ngôn ngữ, nhờ vào khả năng xử lý dữ liệu trừu tượng và cấu trúc cây cú pháp (Abstract Syntax Tree – AST).

**2.2 Phân tích ngữ pháp của ngôn ngữ Basic**

**1. Các định nghĩa cơ bản**

* **Hằng số (Constants)** :
  + Số nguyên (**IntegerLiteral**):

IntegerLiteral ::= [0-9]+

* + Số thực (**RealLiteral**):

RealLiteral ::= [0-9]+"."([0-9]\*)?(e[-+]?[0-9]+)?

* + Chuỗi (**StringLiteral**):

StringLiteral ::= "\""[^\"]\*"\""

* **Tên biến hoặc hàm (Identifier)** :

Identifier ::= [A-Za-z][A-Za-z0-9\_]\*

* **Các từ khóa** :
  + **Procedure**, **Routine**, **Finish**, **Define**, **Allocate**, **Display**, **Condition**, **Clause**, **Repeat**, **Terminate**.
* **Toán tử so sánh (ComparisonOperator)** :

ComparisonOperator ::= > | < | >= | <= | = | <>

* **Toán tử số học (ArithmeticOperator)** :
  + Phép cộng/trừ: **+**, **-**.
  + Phép nhân/chia: **\***, **/**.

**2. Cấu trúc chương trình**

* Một chương trình (**ProgramStructure**):

ProgramStructure ::= GlobalDeclarations

GlobalDeclarations ::= Declaration GlobalDeclarations | eps

* **Các khai báo toàn cục (Declaration)** :

Declaration ::= ProcedureDeclaration | ProcedureDefinition | FunctionDeclaration | FunctionDefinition | VariableDeclaration

**3. Khai báo thủ tục và hàm**

* **Khai báo thủ tục (ProcedurePrototype)** :

ProcedurePrototype ::= Procedure Identifier "(" ParameterList ")"

* **Khai báo hàm (FunctionPrototype)** :

FunctionPrototype ::= Routine VariableName "(" ParameterList ")"

* **Định nghĩa thủ tục (ProcedureDefinition)** :

ProcedureDefinition ::= ProcedurePrototype StatementBlock Finish Procedure

* **Định nghĩa hàm (FunctionDefinition)** :

FunctionDefinition ::= FunctionPrototype StatementBlock Finish Routine

* **Khai báo trước thủ tục/hàm (ProcedurePreDeclaration và FunctionPreDeclaration)** :

ProcedurePreDeclaration ::= Define ProcedurePrototype

FunctionPreDeclaration ::= Define FunctionPrototype

**4. Khai báo biến**

* **Khai báo biến (VariableDeclaration)** :

VariableDeclaration ::= Allocate VariableOrArray Initialization

Initialization ::= "=" Expression | "=" InitialValueList | eps

* **Khởi tạo giá trị mảng (InitialValueList)** :

InitialValueList ::= "{" ValueList "}"

ValueList ::= Value "," ValueList | Value

Value ::= Expression | InitialValueList

**5. Danh sách tham số và đối số**

* **Danh sách tham số (ParameterList)** :

ParameterList ::= VariableOrArray "," ParameterList | VariableOrArray | eps

* **Danh sách đối số (ArgumentList)** :

ArgumentList ::= Expression "," NonEmptyArgumentList | Expression | eps

**6. Các câu lệnh**

* **Câu lệnh (Statement)** :

Statement ::= VariableDeclaration | Assignment | FunctionCallOrArrayAccess | DisplayCall | Loop | Conditional

* **Gán giá trị (Assignment)** :

Assignment ::= VariableName "=" Expression | FunctionCallOrArrayAccess "=" Expression

* **Điều kiện (Conditional)** :

Conditional ::= Condition Expression Clause StatementBlock ElseClause Finish Condition

ElseClause ::= Clause StatementBlock | eps

* **Vòng lặp (Loop)** :

ForLoop ::= Repeat VariableName "=" Expression To Expression StatementBlock Next VariableName

WhileLoop ::= Do PreOrPostLoop

PreOrPostLoop ::= While PreLoop | Until PreLoop | PostLoop

**7. Biểu thức**

* **Biểu thức (Expression)** :

Expression ::= ArithmeticExpression | ArithmeticExpression ComparisonOperator ArithmeticExpression

* **Biểu thức số học (ArithmeticExpression)** :

ArithmeticExpression ::= Term | ArithmeticExpression ArithmeticOperator Term

Term ::= Factor | Term "\*" Factor | Term "/" Factor

Factor ::= VariableName | Literal | "(" Expression ")" | FunctionCallOrArrayAccess

2.3 Phân tích cú pháp và từ vựng của ngôn ngữ

Để hỗ trợ việc xây dựng trình phân tích cú pháp (parser), thư viện **parser\_edsl.py** đã được sử dụng. Thư viện này hỗ trợ cả phân tích từ vựng (lexical analysis) và phân tích cú pháp (syntax analysis), giúp chuyển đổi mã nguồn thành cây cú pháp trừu tượng (AST) một cách hiệu quả. Một ưu điểm nổi bật của **parser\_edsl.py** là khả năng tự động kiểm tra xem ngữ pháp có thuộc lớp LALR(1) hay không, đảm bảo rằng trình biên dịch có thể xử lý các quy tắc ngữ pháp phức tạp mà không gặp lỗi. Ngoài ra, thư viện này còn cung cấp các công cụ để xử lý lỗi cú pháp và tạo báo cáo chi tiết, giúp người phát triển dễ dàng gỡ lỗi và tối ưu hóa trình biên dịch.

Trong quá trình phân tích cú pháp, việc tính toán các thuộc tính của terminal và ký hiệu không đầu cuối đóng vai trò quan trọng. Thư viện **parser\_edsl.py** cung cấp cơ chế để thực hiện điều này một cách hiệu quả:

1. Tính toán thuộc tính của ký hiệu đầu cuối

2. Đánh giá thuộc tính của ký hiệu không kết thúc

3. Xử lý ngoại lệ và lỗi

4. Xác định cấu trúc cú pháp

Bằng cách kết hợp các quy tắc ngữ pháp và các hàm tạo cấu trúc cú pháp, thư viện parser\_edsl.py cung cấp một cơ chế mạnh mẽ để xây dựng một trình phân tích cú pháp hiệu quả. Việc tính toán thuộc tính của terminal và ký hiệu không kết thúc, cùng với khả năng xử lý lỗi và tạo báo cáo chi tiết, giúp quá trình phát triển trình biên dịch trở nên thuận lợi và đáng tin cậy

**Kết quả sẽ có dạng như sau:**

**sadsdfsdffdsfsdf**

2.3 Phân tích ngữ nghĩa

Sau khi xây dựng cây cú pháp, cần tiến hành kiểm tra từng cấu trúc cú pháp để đảm bảo rằng chúng tuân thủ các quy tắc ngữ nghĩa đã được định nghĩa trong phần mô tả ngôn ngữ. Quá trình này bao gồm việc xác minh tính đúng đắn của các cấu trúc cú pháp và loại bỏ sự mơ hồ giữa các lệnh gọi hàm và truy cập mảng.

**Kiểm tra Ngữ Nghĩa cho Cấu Trúc Cú Pháp**

1. **FunctionProto** :
   * Đảm bảo rằng tên tham số và tên hàm là duy nhất.
2. **FunctionDecl và FunctionDef** :
   * Kiểm tra tính duy nhất của chữ ký hàm hoặc coroutine trong phạm vi toàn cục.
3. **VariableDecl** :
   * Đảm bảo rằng tên biến là duy nhất trong phạm vi cục bộ và quá trình khởi tạo (bằng biểu thức hoặc danh sách khởi tạo) được thực hiện chính xác.
4. **FuncCallOrArrayIndex** :
   * Xác nhận rằng các lệnh gọi hàm hoặc lập chỉ mục mảng tồn tại và được sử dụng đúng cách.
5. **FuncCall** :
   * Kiểm tra tính hợp lệ của các lệnh gọi hàm, bao gồm số lượng và kiểu của các đối số.
6. **ExitFor** :
   * Đảm bảo rằng biến được chỉ định tồn tại và nằm ngoài khối vòng lặp có bộ đếm.
7. **ExitWhile, ExitSubroutine, ExitFunction** :
   * Xác minh rằng các lệnh thoát được sử dụng bên ngoài các khối vòng lặp không có bộ đếm hoặc các hàm/coroutine.
8. **AssignStatement** :
   * Kiểm tra tính đúng đắn của phép gán giá trị cho biến.
9. **ForLoop** :
   * Đảm bảo rằng tên bộ đếm là duy nhất, giá trị ban đầu và cuối cùng là số nguyên, và bước tăng/giảm phù hợp với kiểu dữ liệu của bộ đếm.
10. **WhileLoop** :
    * Kiểm tra tính toàn vẹn của điều kiện vòng lặp nếu nó được cung cấp.
11. **IfElseStatement** :
    * Đảm bảo rằng điều kiện là một giá trị số nguyên.
12. **UnaryOpExpr** :
    * Kiểm tra tính hợp lệ của các phép toán đơn nguyên khi áp dụng cho biến.
13. **BinOpExpr** :
    * Đảm bảo rằng các phép toán nhị phân được áp dụng trên các kiểu dữ liệu tương thích. Nếu các kiểu dữ liệu không đồng nhất, cần kiểm tra khả năng chuyển đổi sang kiểu chung.

**Xử lý Lỗi Ngữ Nghĩa**

Trong quá trình phân tích ngữ nghĩa, nhiều lỗi có thể phát sinh do các vấn đề về cú pháp hoặc ngữ nghĩa. Để xử lý các lỗi này, hệ thống đã triển khai các lớp lỗi cụ thể với thông báo rõ ràng và xây dựng bảng ký hiệu để quản lý phạm vi và kiểu dữ liệu. Một số lỗi phổ biến bao gồm:

* **NegativeSizeError** : Kích thước mảng không thể âm.
* **NonConstSizeInitializationError** : Kích thước mảng không được xác định trong quá trình khởi tạo danh sách.
* **LengthMismatchError** : Không khớp giữa kích thước mảng và danh sách khởi tạo.
* **LaunchizationTypeError** : Mảng được khởi tạo bằng một biểu thức.
* **InitializerListDimensionMismatchError** : Kích thước danh sách khởi tạo không khớp.
* **InvalidInitializerListError** : Phát hiện cả biểu thức và danh sách khởi tạo trong danh sách khởi tạo.
* **ConversionError** : Không thể chuyển đổi kiểu dữ liệu.
* **InvalidExitError** : Lệnh thoát không hợp lệ.
* **UndefinedFunctionError** : Hàm chưa được định nghĩa.
* **UndefinedSymbolError** : Ký hiệu chưa được khai báo.
* **RedefinedSymbolError** : Ký hiệu đã được khai báo trước đó.
* **UnexpectedNextForError** : Bộ đếm không được chỉ định trong vòng lặp For.
* **NotIntegerCounterError** : Bộ đếm vòng lặp không phải là kiểu số nguyên.
* **NonIntegerArraySizeInitializationError** : Kích thước mảng không phải là số nguyên trong quá trình khởi tạo.
* **NonIntegerConditionError** : Điều kiện vòng lặp hoặc câu lệnh If không phải là số nguyên.
* **ArrayIndexingDimensionMismatchError** : Số lượng chỉ mục vượt quá kích thước mảng.
* **NonIntegerArrayIndexingError** : Chỉ mục mảng không phải là số nguyên.
* **IncompatibleBinaryTypesError** : Các kiểu dữ liệu không tương thích trong phép toán nhị phân.
* **InvalidUnaryTypeError** : Phép toán đơn nguyên không hợp lệ với kiểu dữ liệu hiện tại.

Bằng cách kết hợp các quy tắc kiểm tra ngữ nghĩa và xử lý lỗi, hệ thống đảm bảo rằng chương trình BASIC được biên dịch sẽ tuân thủ tất cả các quy tắc ngữ nghĩa và loại bỏ các lỗi tiềm ẩn. Điều này giúp nâng cao độ tin cậy và hiệu quả của trình biên dịch.

**2.4 Tạo mã trung gian**

**Mục tiêu**

Giai đoạn tạo mã trung gian chuyển đổi cây cú pháp trừu tượng (AST) thành mã LLVM-IR, một dạng biểu diễn trung gian cấp thấp. Mã này có thể được tối ưu hóa và biên dịch tiếp thành mã máy cho nền tảng đích cụ thể.

**Quy trình thực hiện**

1. **Chọn kiến trúc mục tiêu** :
   * Xác định bộ xử lý, hệ điều hành và các đặc điểm khác của nền tảng đích.
   * Điều này giúp đảm bảo rằng mã trung gian được tạo ra tương thích với môi trường thực thi cuối cùng.
2. **Chuyển đổi AST thành mã trung gian** :
   * Dựa trên cấu trúc của AST, từng nút trong cây được ánh xạ thành các lệnh tương ứng trong LLVM-IR.
   * Ví dụ:
     + Các phép toán số học được dịch thành các lệnh như **add**, **sub**, **mul**, **div**.
     + Các câu lệnh điều khiển như vòng lặp hoặc điều kiện được dịch thành các khối lệnh có điều kiện (**br**) và nhãn (**label**).
3. **Tạo dữ liệu bổ sung** :
   * Ngoài mã lệnh, giai đoạn này cũng tạo ra các cấu trúc dữ liệu cần thiết, chẳng hạn như bảng dữ liệu, bảng bộ nhớ ảo hoặc bảng nhảy.
4. **Hỗ trợ môi trường thời gian chạy** :
   * Mã trung gian có thể bao gồm các lệnh gọi đến các hàm thư viện hỗ trợ thời gian chạy (runtime library), ví dụ: in dữ liệu ra màn hình hoặc xử lý chuỗi.
5. **Ví dụ về mã trung gian** :
   * Một khai báo biến đơn giản trong BASIC:

basic

Copy

1

i% = 0

Được dịch thành LLVM-IR:

llvm

Copy

1

2

%iI = alloca i32, align 4

store i32 0, i32\* %iI, align 4

1. **Kết quả** :
   * Mã trung gian được tạo ra dưới dạng tệp **.ll** (LLVM-IR). Tệp này sẽ được sử dụng trong các giai đoạn tiếp theo như tối ưu hóa và liên kết.

**2. 5 Thư viện hỗ trợ thời gian chạy**

**Mục tiêu**

Thư viện hỗ trợ thời gian chạy cung cấp các chức năng cơ bản để thực thi chương trình, chẳng hạn như in dữ liệu, xử lý chuỗi và quản lý bộ nhớ. Thư viện này được viết bằng ngôn ngữ C và liên kết với mã trung gian LLVM-IR.

**Các chức năng chính**

1. **Xuất dữ liệu** :
   * Các hàm như **PrintI**, **PrintL**, **PrintD**, **PrintF** và **PrintS** được triển khai để xuất các kiểu dữ liệu khác nhau (số nguyên, số thực, chuỗi).
   * Ví dụ:

extern "C" void PrintI(int val) { printf("%d", val); }

extern "C" void PrintS(wchar\_t\* val) { printf("%ls", val); }

1. **Xử lý chuỗi** :
   * Hàm **StringConcat** nối hai chuỗi ký tự.
   * Hàm **StringCopy** sao chép chuỗi từ nguồn sang đích.
   * Ví dụ:

c

Copy

1

2

3

4

5

⌄

extern "C" wchar\_t\* StringConcat(wchar\_t\* lhs, wchar\_t\* rhs) {

unsigned long \_\_sz\_lhs = wcslen(lhs), \_\_sz\_rhs = wcslen(rhs);

lhs = (wchar\_t\*)reallocarray(lhs, \_\_sz\_lhs + \_\_sz\_rhs + 1, sizeof(wchar\_t));

return wcscat(lhs, rhs);

}

1. **Điểm vào chương trình** :

Hàm **Main** được định nghĩa là điểm vào của chương trình. Nó chịu trách nhiệm khởi tạo và điều phối quá trình thực thi.

1. **Liên kết với mã trung gian** :
   * Sử dụng công cụ **llvm-link**, mã trung gian LLVM-IR được liên kết với thư viện hỗ trợ thời gian chạy và các mô-đun khác để tạo ra một mô-đun hoàn chỉnh.

**2.6 Tối ưu hóa**

Giai đoạn tối ưu hóa cải thiện hiệu suất và kích thước của mã trung gian LLVM-IR trước khi biên dịch thành mã máy.

Công cụ **opt** trong hệ sinh thái LLVM được sử dụng để áp dụng các kỹ thuật tối ưu hóa.

3. Kết quả