BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC PHENIKAA**



**BÁO CÁO BÀI TẬP LỚN**

**HỌC PHẦN CHƯƠNG TRÌNH DỊCH**

**BTL**

|  |  |
| --- | --- |
| **Nguyễn Phi Hùng**  **Trương Việt Anh**  **Quân** | 21010598 |
| **Giảng viên hướng dẫn** |  |
| **Tên lớp:** |  |

**Hà Nội, Năm 2024**

MỤC LỤC

[1. Giới thiệu 5](#_Toc160479455)

[1.1. Giới thiệu về chung 5](#_Toc160479456)

[1.2. Mục tiêu 5](#_Toc160479457)

[2. Chương trình dịch 5](#_Toc160479458)

[2.1. Khái niệm chương trình dịch 5](#_Toc160479459)

[2.2. Khái niệm về Token 5](#_Toc160479460)

[2.3. Khái niệm về Phân tích từ vừng 6](#_Toc160479461)

[3. Nội dung 7](#_Toc160479462)

[3.1. Bài 1 7](#_Toc160479463)

[3.1.1. Phân loại các Token trong ngôn ngữ lập trình C 7](#_Toc160479464)

[3.1.2. Thuật toán 7](#_Toc160479465)

[3.1.3. Lập trình 7](#_Toc160479466)

[3.1.4. Phân tích 7](#_Toc160479467)

[3.2. Bài 2 7](#_Toc160479468)

[3.2.1. JSON là gì? 7](#_Toc160479469)

[3.2.2. Lập trình 7](#_Toc160479470)

[4. Kết luận 7](#_Toc160479471)

[4.1. 7](#_Toc160479472)

[4.2. 7](#_Toc160479473)

[5. 7](#_Toc160479474)

[5.1. 7](#_Toc160479475)

[5.2. 7](#_Toc160479476)

[5.2.1. 7](#_Toc160479477)

[5.2.2. 7](#_Toc160479478)

[5.2.3. 7](#_Toc160479479)

[6. Thử nghiệm và đánh giá 7](#_Toc160479480)

[6.1. 7](#_Toc160479481)

[7. Kêt luận 7](#_Toc160479482)

[7.1. Tóm tắt những gì đã đạt được: 7](#_Toc160479483)

[7.2. Đề xuất 7](#_Toc160479484)

[TÀI LIỆU THAM KHẢO 7](#_Toc160479485)

[Phụ lục 9](#_Toc160479486)

**BẢNG BIỂU VÀ HÌNH ẢNH**

# Giới thiệu

## Giới thiệu chung

**Câu 1: Phân loại Tokens trong Ngôn Ngữ C**

Trong phần này, chúng ta sẽ tìm hiểu cách phân loại các token trong ngôn ngữ lập trình C và đề xuất một thuật toán để tạo ra một bộ phân tích từ vựng hiệu quả. Thuật toán sẽ được giải thích và phân tích sâu hơn để hiểu tại sao nó được chọn. Sau đó, chúng ta sẽ thực hiện việc triển khai thuật toán này vào mã nguồn.

**Câu 2: Phân Tích Từ Vựng cho JSON**

Ở phần này, chúng ta sẽ xây dựng một bộ phân tích từ vựng cho JSON (JavaScript Object Notation), một định dạng phổ biến trong truyền thông dữ liệu giữa các ứng dụng web. Chúng ta sẽ tìm hiểu về cấu trúc của JSON và sau đó xây dựng một bộ phân tích từ vựng để nhận dạng các thành phần của nó.

## Mục tiêu

Mục tiêu của cả hai bài tập là nắm vững quy trình phân tích từ vựng và triển khai thuật toán để xây dựng bộ phân tích từ vựng cho các ngôn ngữ lập trình khác nhau, từ đó nâng cao hiểu biết và kỹ năng trong lĩnh vực này.

# Chương trình dịch

## Khái niệm chương trình dịch

Chương trình dịch – hay còn được gọi với tên tiếng Anh là compiler, là chương trình có chức năng chuyển đổi chương trình nguồn được viết bằng ngôn ngữ lập trình bậc cao sang chương trình đích được thể hiện bằng ngôn ngữ máy và chương trình đích này có thể chạy (thực thi) trên máy tính được. Vì ngôn ngữ lập trình bậc cao không thể nạp trực tiếp vào bộ nhớ và thực hiện ngay như mã máy nên cần chương trình dịch để chuyển đổi chương trình viết bằng ngôn ngữ lập trình bậc cao sang mã máy.

Một chương trình dịch chịu trách nhiệm dịch một chuỗi các hướng dẫn được viết bằng một ngôn ngữ lập trình cụ thể (tức là ngôn ngữ nguồn hoặc mã nguồn) sang một chương trình mới nhưng ở dạng ngôn ngữ máy tính (ngôn ngữ đích).

Nói chung, ngôn ngữ đích là ngôn ngữ cấp thấp hơn được sử dụng để máy tính có thể hiểu các hướng dẫn bằng văn bản. Ngôn ngữ duy nhất máy có thể trực tiếp hiểu và thực hiện. Trình biên dịch tạo ra một chương trình mới còn được gọi là mã đối tượng. Trong khi đó, Ngôn ngữ bậc cao rất gần với ngôn ngữ tự nhiên, có tính độc lập cao, ít phụ thuộc vào loại máy và chương trình phải dịch sang ngôn ngữ máy mới thực hiện được

Hầu hết các trình biên dịch sẽ dịch mã nguồn được viết bằng ngôn ngữ cấp cao sang mã đối tượng hoặc ngôn ngữ máy để được thực thi trực tiếp bởi máy tính hoặc máy ảo. Tuy nhiên, cũng có trường hợp chương trình dịch có khả năng dịch từ ngôn ngữ cấp thấp sang ngôn ngữ cấp cao. Trình biên dịch như vậy được gọi là dịch ngược. Đồng thời cũng sẽ có các chương trình dịch từ ngôn ngữ cấp cao này sang ngôn ngữ cấp cao khác.​

## Khái niệm về Từ tố

Trong lĩnh vực xử lý ngôn ngữ tự nhiên, từ tố - những đơn vị nhỏ nhất của ngôn ngữ - đóng vai trò quan trọng, là chìa khóa mở cánh cửa cho sự hiểu biết sâu sắc về ý nghĩa và ngữ cảnh.

"Từ tố" trong ngữ cảnh của chương trình dịch thường được hiểu là "token" trong tiếng Anh. Trong lĩnh vực lập trình và xử lý ngôn ngữ tự nhiên, một "từ tố" là đơn vị nhỏ nhất của ngôn ngữ, có thể là một từ, một ký tự, hoặc một phần của từ. Trong ngữ cảnh của chương trình dịch, "từ tố" thường là các đơn vị nhỏ nhất được xử lý để dịch từ một ngôn ngữ sang ngôn ngữ khác. Đối với các mô hình dịch ngôn ngữ tự nhiên, các từ tố có thể là từ, cụm từ, hoặc thậm chí là từng phần của câu. Ví dụ: hằng, biến, từ khoá, các phép toán,…

## Khái niệm về Phân tích từ vừng

Phân Tích Từ Vựng, còn được gọi là phân tích từ vựng (lexical analysis) hoặc tokenization, là quá trình đầu tiên trong quá trình biên dịch hoặc dịch mã nguồn. Trong quá trình này, chuỗi ký tự đầu vào từ mã nguồn sẽ được chia thành các đơn vị nhỏ hơn gọi là "token". Mỗi token đại diện cho một phần của ngôn ngữ lập trình, bao gồm từ khóa (keywords), biến, hằng số, toán tử, và các ký tự đặc biệt.

Quá trình phân tích từ vựng thường bắt đầu bằng việc đọc chuỗi ký tự từ mã nguồn một cách tuần tự. Khi gặp một chuỗi ký tự, trình biên dịch hoặc chương trình dịch sẽ so sánh nó với các quy tắc cú pháp được định nghĩa trước đó để xác định xem chuỗi đó có phải là một token và loại token nào.

Các bước cơ bản trong quá trình phân tích từ vựng bao gồm:

1. **Tokenization:** Chia chuỗi ký tự thành các token nhỏ dựa trên các quy tắc cú pháp của ngôn ngữ lập trình.
2. **Kiểm Tra Từ Khóa:** So sánh các token với danh sách từ khóa được xác định trước để xác định xem chúng có phải là từ khóa hay không.
3. **Nhận Dạng Biến và Hằng Số:** Xác định các token là biến hoặc hằng số và gán các giá trị tương ứng nếu cần.
4. **Xử Lý Ký Tự Đặc Biệt:** Xử lý các ký tự đặc biệt như dấu ngoặc, dấu phẩy, dấu chấm phẩy, v.v.

Kết quả của quá trình phân tích từ vựng là một danh sách các token đã được nhận dạng và chuyển đổi từ chuỗi ký tự của mã nguồn. Các token này sẽ được sử dụng cho quá trình phân tích cú pháp và các bước tiếp theo trong quá trình biên dịch hoặc dịch mã nguồn.

# Nội dung

## Bài 1

**Nội dung:** Phân loại các tokens của ngôn ngữ C. Hãy đề xuất một thuật toán để tạo ra một bộ phân tích tự vựng. Minh họa bằng một chương trình viết bằng ngôn ngữ C/C++/Java hoặc Python.

### Phân loại các Token trong ngôn ngữ lập trình C

Ngôn ngữ lập trình C có nhiều loại token khác nhau, mỗi token có thể là một từ khóa, một định danh, biến, hằng số, ký tự có nghĩa trong lập trình C, mỗi loại đều có chức năng và ý nghĩa riêng. Một số loại token cơ bản trong ngôn ngữ C:

* Identifiers (Tên biến và hàm): Được sử dụng để đặt tên cho biến, hàm, hoặc các thành phần khác trong chương trình. Phải bắt đầu bằng một chữ cái hoặc dấu gạch dưới (\_) và có thể chứa chữ cái, chữ số và dấu gạch dưới.
* Ví dụ:

|  |
| --- |
| int myVariable;  void calculateSum(); |

* Keywords (Từ khóa): Các từ có ý nghĩa đặc biệt trong ngôn ngữ C. Không thể sử dụng làm tên cho biến hoặc hàm. Ví dụ: if, else, for, while, int, char, return,...
* Constants (Hằng số): Các giá trị không thay đổi trong suốt chương trình. Có thể là hằng số số học (integer, floating-point) hoặc chuỗi.
* Ví dụ:

|  |
| --- |
| const int MAX\_SIZE = 100;  float pi = 3.14;  char greeting[] = "Hello"; |

* String literals (Chuỗi ký tự): Chuỗi ký tự được đặt trong dấu ngoặc kép.
* Ví dụ:

|  |
| --- |
| "Hello, World!", "C programming" |

* Operators (Toán tử): Các ký tự hoặc từ có chức năng thực hiện các phép toán. Ví dụ: +, -, \*, /, %, ==, !=, >, <, <=, >=, &&, ||, !, +=, -=,...
* Punctuation (Dấu câu): Ký tự dấu câu được sử dụng để phân cách và kết thúc các phần trong mã nguồn. Ví dụ: ; (dấu chấm phẩy), , (dấu phẩy), . (dấu chấm), : (dấu hai chấm), () (dấu ngoặc đơn), {} (dấu ngoặc nhọn), [] (dấu ngoặc vuông),...
* Separators (Dấu phân cách): Ký tự phân cách được sử dụng để phân chia các phần của mã nguồn. Ví dụ: ( và ) để đánh dấu đầu vào và đầu ra của hàm.
* Comments (Chú thích): Giải thích mã nguồn và không được biên dịch. Có hai loại chú thích: // cho chú thích trên một dòng và /\* \*/ cho chú thích trên nhiều dòng.
* Directive (Chỉ thị tiền xử lý): Bắt đầu bằng dấu #, định rõ các chỉ thị tiền xử lý cho trình biên dịch.
* Ví dụ:

|  |
| --- |
| #include <stdio.h>, #define MAX\_SIZE 100 |

### Chương trình và giải thích

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h> |

Giải thích

Thư viện **stdio.h** cung cấp các hàm và định nghĩa liên quan đến nhập và xuất dữ liệu từ và đến các luồng (streams), chẳng hạn như nhập từ bàn phím và xuất ra màn hình.

Thư viện **stdlib.h** cung cấp các hàm tiện ích phổ biến như quản lý bộ nhớ động, quản lý chuỗi ký tự và các hàm toán học.

Thư viện **string.h** cung cấp các hàm và định nghĩa để thao tác với chuỗi ký tự, chẳng hạn như sao chép chuỗi, so sánh chuỗi và tìm kiếm trong chuỗi.

Thư viện **ctype.h** cung cấp các hàm để kiểm tra và chuyển đổi các ký tự, như kiểm tra xem một ký tự có phải là chữ cái hay không, chuyển đổi chữ hoa thành chữ thường và ngược lại.

|  |
| --- |
| typedef enum  {  START,  IN\_IDENTIFIER,  IN\_NUMBER,  IN\_OPERATOR,  IN\_DELIMITER,  DONE  } State; |

Giải thích:

**typedef enum** được sử dụng để định nghĩa một kiểu dữ liệu mới là một loại liệt kê (enumeration). Liệt kê (enum) là một tập hợp các hằng số được gán các giá trị nguyên liên tục.

Trong trường hợp này, **typedef enum** định nghĩa một kiểu liệt kê mới được đặt tên là **State**. Kiểu liệt kê này có các giá trị sau:

**START**: Đại diện cho trạng thái ban đầu của một hành động hoặc quá trình.

**IN\_IDENTIFIER**: Đại diện cho trạng thái khi đang xử lý một phần của biến hoặc từ khóa trong mã nguồn.

**IN\_NUMBER**: Đại diện cho trạng thái khi đang xử lý một phần của một số trong mã nguồn.

**IN\_OPERATOR**: Đại diện cho trạng thái khi đang xử lý một phần của một toán tử trong mã nguồn.

**IN\_DELIMITER**: Đại diện cho trạng thái khi đang xử lý một phần của một dấu phân cách trong mã nguồn.

**DONE**: Đại diện cho trạng thái kết thúc của một hành động hoặc quá trình.

|  |
| --- |
| #define KEYWORD "Keyword"  #define IDENTIFIER "Identifier"  #define OPERATOR "Operator"  #define LITERAL "Literal"  #define DELIMITER "Delimiter" |

Giải thích:

**#define** sau định nghĩa các hằng số được sử dụng để đại diện cho các loại token khác nhau trong quá trình phân tích mã nguồn:

**KEYWORD**: Được định nghĩa là chuỗi **"Keyword"**. Hằng số này được sử dụng để đại diện cho các từ khóa trong ngôn ngữ lập trình C như **if**, **else**, **while**, v.v.

**IDENTIFIER**: Được định nghĩa là chuỗi **"Identifier"**. Hằng số này được sử dụng để đại diện cho các biến hoặc tên được đặt trong mã nguồn.

**OPERATOR**: Được định nghĩa là chuỗi **"Operator"**. Hằng số này được sử dụng để đại diện cho các toán tử trong ngôn ngữ lập trình C như **+**, **-**, **\***, v.v.

**LITERAL**: Được định nghĩa là chuỗi **"Literal"**. Hằng số này được sử dụng để đại diện cho các giá trị chữ và số cố định trong mã nguồn.

**DELIMITER**: Được định nghĩa là chuỗi **"Delimiter"**. Hằng số này được sử dụng để đại diện cho các dấu phân cách hoặc dấu câu như **()**, **[]**, **{}**, v.v.

|  |
| --- |
| int is\_operator(char c)  {  const char operators[] = "+-\*/%=&|><!^~";  return strchr(operators, c) != NULL;  } |

Giải thích:

Hàm **is\_operator** được viết để kiểm tra xem một ký tự có phải là một toán tử trong ngôn ngữ lập trình C hay không. Đầu vào của hàm là một ký tự **c** cần được kiểm tra.

Bên trong hàm, một mảng ký tự **operators[]** được khai báo và khởi tạo với tập hợp các ký tự đại diện cho các toán tử trong ngôn ngữ lập trình C, bao gồm **+**, **-**, **\***, **/**, **%**, **=**, **&**, **|**, **>**, **<**, **!**, **^**, và **~**.

Sau đó, hàm sử dụng hàm **strchr()** trong thư viện **string.h** để kiểm tra xem ký tự **c** có xuất hiện trong mảng **operators[]** hay không. Hàm **strchr()** sẽ trả về con trỏ tới vị trí đầu tiên của ký tự **c** trong chuỗi nếu nó được tìm thấy, và **NULL** nếu không tìm thấy.

Cuối cùng, kết quả của biểu thức **strchr(operators, c) != NULL** sẽ trả về **1** nếu ký tự **c** là một toán tử, và **0** nếu không phải. Kết quả này được trả về từ hàm **is\_operator**.

|  |
| --- |
| int is\_delimiter(char c)  {  const char delimiters[] = "()[]{};,:";  return strchr(delimiters, c) != NULL;  } |

Hàm **is\_delimiter** được viết để kiểm tra xem một ký tự **c** có phải là một dấu phân cách (delimiter) trong ngôn ngữ lập trình C hay không.

Trong hàm này, một mảng ký tự **delimiters[]** được khai báo và khởi tạo với một tập hợp các ký tự đại diện cho các dấu phân cách thường được sử dụng trong ngôn ngữ lập trình C, bao gồm: **()**, **[]**, **{}**, **;**, **,**, và **:**.

Tiếp theo, hàm sử dụng hàm **strchr()** từ thư viện **string.h** để kiểm tra xem ký tự **c** có xuất hiện trong mảng **delimiters[]** hay không. Hàm **strchr()** sẽ trả về con trỏ tới vị trí đầu tiên của ký tự **c** trong chuỗi nếu nó được tìm thấy, và **NULL** nếu không tìm thấy.

Kết quả cuối cùng của biểu thức **strchr(delimiters, c) != NULL** sẽ trả về **1** nếu ký tự **c** là một dấu phân cách, và **0** nếu không phải. Giá trị này được trả về từ hàm **is\_delimiter**.

|  |
| --- |
| int is\_keyword(const char \*word)  {  const char \*keywords[] = {"auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register", "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while"};  int num\_keywords = sizeof(keywords) / sizeof(keywords[0]);  for (int i = 0; i < num\_keywords; ++i)  {  if (strcmp(word, keywords[i]) == 0)  {  return 1;  }  }  return 0;  } |

Giải thích:

Hàm **is\_keyword** được viết để kiểm tra xem một từ (biến trỏ bởi con trỏ **word**) có phải là một từ khóa trong ngôn ngữ lập trình C hay không.

Trong hàm này:

1. Một mảng con trỏ **keywords[]** được khai báo và khởi tạo với một tập hợp các từ khóa trong ngôn ngữ lập trình C, bao gồm tất cả các từ khóa chuẩn như **auto**, **break**, **case**, **char**, **const**, v.v.
2. Biến **num\_keywords** được khởi tạo để lưu trữ số lượng các từ khóa trong mảng **keywords[]**. Để tính số lượng này, chia kích thước của mảng **keywords[]** cho kích thước của phần tử đầu tiên của mảng **keywords[]** (do mỗi phần tử trong **keywords[]** là một con trỏ).
3. Một vòng lặp **for** được sử dụng để duyệt qua từng từ khóa trong mảng **keywords[]**. Trong mỗi lần lặp, hàm **strcmp()** được sử dụng để so sánh từ **word** với từ khóa hiện tại trong mảng. Nếu từ **word** giống với từ khóa nào đó trong mảng, hàm **strcmp()** sẽ trả về 0, và hàm **is\_keyword** sẽ trả về **1** để báo hiệu rằng từ **word** là một từ khóa.
4. Nếu không có từ khóa nào trùng khớp với từ **word**, vòng lặp sẽ kết thúc và hàm sẽ trả về **0** để báo hiệu rằng từ **word** không phải là một từ khóa.

|  |
| --- |
| // Hàm chính để phân tích một đoạn mã và trả về các token  void tokenize(const char \*code)  {  State state = START;  char lexeme[MAX\_TOKEN\_LENGTH];  int lexeme\_index = 0;  for (int i = 0; code[i] != '\0'; ++i)  {  char current\_char = code[i];  switch (state)  {  case START:  if (isalpha(current\_char) || current\_char == '\_')  {  state = IN\_IDENTIFIER;  lexeme[lexeme\_index++] = current\_char;  }  else if (isdigit(current\_char))  {  state = IN\_NUMBER;  lexeme[lexeme\_index++] = current\_char;  }  else if (is\_operator(current\_char))  {  state = IN\_OPERATOR;  lexeme[lexeme\_index++] = current\_char;  }  else if (is\_delimiter(current\_char))  {  state = IN\_DELIMITER;  lexeme[lexeme\_index++] = current\_char;  }  break;  case IN\_IDENTIFIER:  if (isalnum(current\_char) || current\_char == '\_')  {  lexeme[lexeme\_index++] = current\_char;  }  else  {  lexeme[lexeme\_index] = '\0';  if (is\_keyword(lexeme))  {  printf("Keyword: %s\n", lexeme);  }  else  {  printf("Identifier: %s\n", lexeme);  }  lexeme\_index = 0;  state = START;  --i; // Re-process current character  }  break;  case IN\_NUMBER:  if (isdigit(current\_char))  {  lexeme[lexeme\_index++] = current\_char;  }  else  {  lexeme[lexeme\_index] = '\0';  printf("Number: %s\n", lexeme);  lexeme\_index = 0;  state = START;  --i; // Re-process current character  }  break;  case IN\_OPERATOR:  if (is\_operator(current\_char))  {  lexeme[lexeme\_index++] = current\_char;  }  else  {  lexeme[lexeme\_index] = '\0';  printf("Operator: %s\n", lexeme);  lexeme\_index = 0;  state = START;  --i; // Re-process current character  }  break;  case IN\_DELIMITER:  if (is\_delimiter(current\_char))  {  lexeme[lexeme\_index++] = current\_char;  }  else  {  lexeme[lexeme\_index] = '\0';  printf("Delimiter: %s\n", lexeme);  lexeme\_index = 0;  state = START;  --i; // Re-process current character  }  break;  case DONE:  // Should never reach here  break;  }  }  // Handling the last token  if (state == IN\_IDENTIFIER)  {  lexeme[lexeme\_index] = '\0';  if (is\_keyword(lexeme))  {  printf("Keyword: %s\n", lexeme);  }  else  {  printf("Identifier: %s\n", lexeme);  }  }  else if (state == IN\_NUMBER)  {  lexeme[lexeme\_index] = '\0';  printf("Number: %s\n", lexeme);  }  else if (state == IN\_OPERATOR)  {  lexeme[lexeme\_index] = '\0';  printf("Operator: %s\n", lexeme);  }  else if (state == IN\_DELIMITER)  {  lexeme[lexeme\_index] = '\0';  printf("Delimiter: %s\n", lexeme);  }  } |

Giải thích:

Hàm **tokenize** được sử dụng để phân tích một đoạn mã nguồn (được truyền vào qua tham số **code**) thành các token và in ra loại của mỗi token.

Hàm bắt đầu bằng việc khởi tạo biến **state** với giá trị ban đầu là **START**, biến **lexeme** là một mảng ký tự để lưu trữ token hiện tại, và biến **lexeme\_index** là chỉ số của mảng **lexeme**.

Sau đó, một vòng lặp **for** được sử dụng để duyệt qua từng ký tự trong đoạn mã nguồn **code** cho đến khi gặp ký tự kết thúc chuỗi (**'\0'**).

Mỗi lần lặp, ký tự hiện tại được lấy ra và gán vào biến **current\_char**. Tiếp theo, một câu lệnh **switch** được sử dụng để xác định trạng thái hiện tại của quá trình phân tích và thực hiện hành động tương ứng:

Trong trạng thái **START**:Nếu ký tự là một chữ cái hoặc ký tự **\_**, chương trình chuyển sang trạng thái **IN\_IDENTIFIER** và thêm ký tự vào **lexeme**.

* Nếu ký tự là một chữ số, chương trình chuyển sang trạng thái **IN\_NUMBER** và thêm ký tự vào **lexeme**.
* Nếu ký tự là một toán tử, chương trình chuyển sang trạng thái **IN\_OPERATOR** và thêm ký tự vào **lexeme**.
* Nếu ký tự là một dấu phân cách, chương trình chuyển sang trạng thái **IN\_DELIMITER** và thêm ký tự vào **lexeme**.

Trong các trạng thái **IN\_IDENTIFIER**, **IN\_NUMBER**, **IN\_OPERATOR**, và **IN\_DELIMITER**, ký tự tiếp theo được thêm vào **lexeme** nếu thỏa mãn điều kiện tương ứng.

Khi kết thúc một token, nếu token đó là một từ khóa thì in ra **"Keyword: "** kèm theo **lexeme**, nếu không, in ra **"Identifier: "** kèm theo **lexeme**. Sau đó, **lexeme\_index** được đặt lại về 0 và **state** được chuyển về **START**. Đồng thời, biến **i** được giảm đi một đơn vị để xử lý lại ký tự hiện tại.

|  |
| --- |
| int main()  {  char code[] = "int main() { return 3; }";  tokenize(code);  return 0;  } |

## Bài 2

Câu 2: Tạo một bộ phân tích từ vựng cho JSON (JavaScript Object Notation: [http://json.org](http://json.org/)). Chương trình nên đọc

JSON làm input, sau đó in ra chuỗi các token quan sát được: LBRACKET, STRING, COLON... Tìm một số chương trình nguồn JSON và kiểm tra bộ Phân tích từ vựng nhóm vừa viết để xem nó có hoạt động không.

### JSON là gì?

**JSON** (Ký hiệu đối tượng JavaScript) là một định dạng trao đổi dữ liệu nhẹ. Nó rất dễ dàng cho con người đọc và viết. Nó rất dễ dàng cho máy móc để phân tích cú pháp và tạo ra. Nó dựa trên một tập hợp con của JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON là một định dạng văn bản hoàn toàn độc lập với ngôn ngữ nhưng sử dụng các quy ước quen thuộc với các lập trình viên thuộc họ ngôn ngữ C, bao gồm C, C ++, C #, Java, JavaScript, Perl, Python và nhiều ngôn ngữ khác. Các thuộc tính này làm cho JSON trở thành một ngôn ngữ trao đổi dữ liệu lý tưởng.

JSON được xây dựng trên hai cấu trúc:

Một tập hợp các cặp tên/giá trị. Trong các ngôn ngữ khác nhau, điều này được thực hiện dưới dạng một *đối tượng*, bản ghi, cấu trúc, từ điển, bảng băm, danh sách khóa hoặc mảng kết hợp.

Một danh sách các giá trị được sắp xếp theo thứ tự. Trong hầu hết các ngôn ngữ, điều này được nhận ra dưới dạng *mảng,* vector, danh sách hoặc chuỗi.

Đây là những cấu trúc dữ liệu phổ quát. Hầu như tất cả các ngôn ngữ lập trình hiện đại đều hỗ trợ chúng dưới dạng này hay dạng khác. Nó có ý nghĩa rằng một định dạng dữ liệu có thể hoán đổi cho nhau với các ngôn ngữ lập trình cũng dựa trên các cấu trúc này.

Trong JSON, chúng có các dạng sau:

Một *đối tượng* là một tập hợp các cặp tên/giá trị không có thứ tự. Một đối tượng bắt đầu  **bằng {***dấu ngoặc nhọn trái* và kết thúc bằng **dấu** *ngoặc nhọn phải*. Mỗi tên được theo sau bởi **: dấu** *hai chấm* và các cặp tên / giá trị được phân tách bằng **dấu** *phẩy*.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

Một *giá trị* có thể là  *một chuỗi* trong dấu ngoặc kép, hoặc *một số, hoặc* true hoặc **false** hoặc null**, hoặc một** đối tượng *hoặc một*  *mảng*. Những cấu trúc này có thể được lồng vào nhau.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, thiết kế

Mô tả được tạo tự động

Một  *chuỗi*  là một chuỗi gồm 0 hoặc nhiều ký tự Unicode, được bọc trong dấu ngoặc kép, sử dụng dấu gạch chéo ngược. Một ký tự được biểu diễn dưới dạng một chuỗi ký tự duy nhất. Một chuỗi rất giống một chuỗi C hoặc Java.

Ảnh có chứa văn bản, ảnh chụp màn hình

Mô tả được tạo tự động

Một *số* rất giống số C hoặc Java, ngoại trừ các định dạng bát phân và thập lục phân không được sử dụng.

Ảnh có chứa ảnh chụp màn hình, văn bản, biểu đồ

Mô tả được tạo tự động

Khoảng trắng có thể được chèn vào giữa bất kỳ cặp mã thông báo nào. Ngoại trừ một vài chi tiết mã hóa, điều đó hoàn toàn mô tả ngôn ngữ.

Ảnh có chứa ảnh chụp màn hình, văn bản, màu đen

Mô tả được tạo tự động

### Lập trình

**Sử dụng ngôn ngữ lập trình Python**

|  |
| --- |
| import re  class JsonLexer:  def \_\_init\_\_(self, text):  self.text = text  self.pos = 0  def next\_token(self):  while self.pos < len(self.text):  c = self.text[self.pos]  if c.isspace():  self.pos += 1  continue  # Kiểm tra các token  if c in "{}[]:,":  self.pos += 1  return c  # Kiểm tra chuỗi  if c == '"':  return self.parse\_string()  # Kiểm tra số  if c.isdigit() or c == '-':  return self.parse\_number()  # Kiểm tra từ khóa  if c.isalpha():  return self.parse\_keyword()  raise ValueError(f"Lỗi cú pháp: {c} tại vị trí {self.pos}")  def parse\_string(self):  start = self.pos + 1  while self.pos < len(self.text):  c = self.text[self.pos]  if c == '"':  self.pos += 1  return self.text[start:self.pos-1]  elif c == '\\':  self.pos += 1  else:  self.pos += 1  raise ValueError("Chuỗi không được đóng")  def parse\_number(self):  start = self.pos  while self.pos < len(self.text):  c = self.text[self.pos]  if not c.isdigit() and c != '.':  break  self.pos += 1  return float(self.text[start:self.pos])  def parse\_keyword(self):  start = self.pos  while self.pos < len(self.text):  c = self.text[self.pos]  if not c.isalnum():  break  self.pos += 1  keyword = self.text[start:self.pos]  if keyword in ["true", "false", "null"]:  return keyword  raise ValueError(f"Từ khóa không hợp lệ: {keyword}") |

**Giải thích chương trình:**

Dòng này import module re cung cấp các biểu thức chính quy để thực hiện so khớp mẫu nâng cao trong Python. Nó không được sử dụng trực tiếp trong mã này, nhưng có thể hữu ích cho các nhiệm vụ phân tích từ vựng phức tạp hơn trong tương lai.

**Định nghĩa lớp: JsonLexer**

|  |
| --- |
| class JsonLexer:  def \_\_init\_\_(self, text):  self.text = text  self.pos = 0  def next\_token(self):  while self.pos < len(self.text):  c = self.text[self.pos]  if c.isspace():  self.pos += 1  continue  # Kiểm tra các token  if c in "{}[]:,":  self.pos += 1  return c  # Kiểm tra chuỗi  if c == '"':  return self.parse\_string()  # Kiểm tra số  if c.isdigit() or c == '-':  return self.parse\_number()  # Kiểm tra từ khóa  if c.isalpha():  return self.parse\_keyword()  raise ValueError(f"Lỗi cú pháp: {c} tại vị trí {self.pos}")  def parse\_string(self):  start = self.pos + 1  while self.pos < len(self.text):  c = self.text[self.pos]  if c == '"':  self.pos += 1  return self.text[start:self.pos-1]  elif c == '\\':  self.pos += 1  else:  self.pos += 1  raise ValueError("Chuỗi không được đóng")  def parse\_number(self):  start = self.pos  while self.pos < len(self.text):  c = self.text[self.pos]  if not c.isdigit() and c != '.':  break  self.pos += 1  return float(self.text[start:self.pos])  def parse\_keyword(self):  start = self.pos  while self.pos < len(self.text):  c = self.text[self.pos]  if not c.isalnum():  break  self.pos += 1  keyword = self.text[start:self.pos]  if keyword in ["true", "false", "null"]:  return keyword  raise ValueError(f"Từ khóa không hợp lệ: {keyword}") |

**Giải thích từng phần:**

***\_\_init\_\_(self, text)*:** Hàm khởi tạo tạo đối tượng với văn bản JSON cần phân tích (text) và đặt vị trí hiện tại (pos) vào đầu văn bản (chỉ mục 0).

***next\_token(self)*:** Đây là phương thức chính của bộ phân tích từ vựng. Nó lặp qua từng ký tự trong văn bản JSON (while self.pos < len(self.text)) và thực hiện các hành động sau:

Bỏ qua các ký tự khoảng trắng (if c.isspace()) bằng cách sử dụng continue.

Nếu ký tự là dấu ngoặc nhọn **{}**, dấu ngoặc vuông **[]**, dấu phẩy **,,** hoặc dấu hai chấm **:**, nó trả về ký tự đó dưới dạng token (if c in "{}[]:,").

Nếu ký tự là dấu ngoặc kép " (biểu thị chuỗi), nó gọi phương thức ***parse\_string***để xử lý token chuỗi.

Nếu ký tự là chữ số hoặc dấu trừ - (biểu thị số), nó gọi phương thức ***parse\_number***để xử lý token số.

Nếu ký tự là chữ cái (biểu thị từ khóa), nó gọi phương thức ***parse\_keyword*** để xử lý token từ khóa.

Nếu không có điều kiện nào trên khớp, nó sẽ ***raise*** một **ValueError** với thông báo lỗi về việc gặp phải ký tự không mong muốn và vị trí của nó.

**Phương thức *parse\_string(self):***

Xử lý các chuỗi thoát (\\) bằng cách tăng vị trí **(self.pos += 1)** để bỏ qua ký tự backslash.

Nếu không tìm thấy dấu ngoặc kép đóng, nó sẽ raise một **ValueError** báo lỗi chuỗi chưa được đóng.

Cuối cùng, nó trả về nội dung chuỗi được trích xuất **(self.text[start:self.pos-1])**.

**Phương thức parse\_number(self):**

Tìm vị trí bắt đầu (start) của số.

Lặp lại cho đến khi gặp ký tự không phải là chữ số hoặc dấu chấm (.).

Trả về giá trị số thực được trích xuất **(float(self.text[start:self.pos]))**.

**Phương thức parse\_keyword(self):**

Tìm vị trí bắt đầu (start) của từ khóa.

Lặp lại cho đến khi gặp ký tự không phải là chữ cái hoặc số.

Trả về từ khóa được trích xuất (keyword) nếu nó là một trong các từ khóa hợp lệ ("true", "false", "null").

Nếu không, raise một **ValueError** báo lỗi từ khóa không hợp lệ.

**Ngoài ra:**

Mã này sử dụng try-except để xử lý các lỗi cú pháp trong JSON.

Có thể mở rộng mã này để hỗ trợ các kiểu dữ liệu JSON khác như mảng và đối tượng.

**Ví dụ sử dụng (Python):**

|  |
| --- |
| def main():  # Ví dụ JSON  text = """  {  "name": "John Doe",  "age": 30,  "address":  {  "street": "123 Main Street",  "city": "New York",  "state": "NY",  "zip": 10001  },  "hobbies":  [  "programming",  "reading",  "hiking"  ]  }  """  lexer = JsonLexer(json\_text)  while True:  token = lexer.next\_token()  if token is None:  break  print(token)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

**Mã này sẽ in ra từng token trong JSON, ví dụ:**

|  |
| --- |
| {  "name  "John Doe"  ,  "age  30  ,  "is\_active  true  ,  "address  {  "street  "123 Main Street"  ,  "city  "New York"  ,  "state  "NY"  ,  "zip  "10001"  } |

**Kiểm tra bộ Phân tích từ vựng**

Đoạn mã sau để kiểm tra bộ phân tích từ vựng với một số chương trình nguồn JSON:

|  |
| --- |
| def test\_lexer(json\_file):  with open(json\_file, "r") as f:  json\_text = f.read()  lexer = JsonLexer(json\_text)  try:  while True:  lexer.next\_token()  except ValueError as e:  print(f"Lỗi: {e}")  if \_\_name\_\_ == "\_\_main\_\_":  test\_lexer("example1.json")  test\_lexer("example2.json") |

# Kết luận

## 

## 

# 

## 

## 

### 

### 

### 

# Thử nghiệm và đánh giá

## 

# Kêt luận

## Tóm tắt những gì đã đạt được:

## Đề xuất

# TÀI LIỆU THAM KHẢO

|  |  |
| --- | --- |
| [1] | AWS, "How to create Bucket S3," [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/creating-bucket.html. |
| [2] | GeeksforGeeks, "What is Elastic Compute Cloud (EC2)?," [Online]. Available: https://www.geeksforgeeks.org/what-is-elastic-compute-cloud-ec2/. |
| [3] | knowledgehut, "knowledgehut," [Online]. Available: https://www.knowledgehut.com/tutorials/aws/creation-of-rds-instance. |
| [4] | git-scm, "Download Git," [Online]. Available: https://git-scm.com/download/linux. |
| [5] | L. Tagliaferri, "How To Install the Anaconda Python Distribution on Ubuntu 20.04," [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-the-anaconda-python-distribution-on-ubuntu-20-04?fbclid=IwAR3uq6htUsJCwevGkyae3vlUvsUcMMkmalmhYCDG9qQKY4pUt6PCkBxGuoE. |

# Phụ lục

Đường dẫn github:

Ứng dụng: https://github.com/fhihung/mushroom\_detection

AI Model: <https://github.com/linhphuong06/AWS.git>