**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA KHOA HỌC MÁY TÍNH**

Logo, company name

Description automatically generated

**BÁO CÁO ĐỒ ÁN MÔN HỌC**

**NGUYÊN LÝ VÀ PHƯƠNG PHÁP LẬP TRÌNH**

**CS111.M21.KHCL**

|  |  |
| --- | --- |
| **SINH VIÊN THỰC HIỆN:** | **PHẠM TRẦN ANH TIÊN - 20522012** |

**TP. HỒ CHÍ MINH, 4/2022**

MỤC LỤC

[1. GIỚI THIỆU 3](#_Toc103102451)

[2. TRÌNH PHÂN TÍCH TỪ VỰNG LEX 3](#_Toc103102452)

[2.1. Mô tả Lex 3](#_Toc103102453)

[2.2. Phân tích từ vựng 3](#_Toc103102454)

[2.3. Cấu trúc của LEX 4](#_Toc103102455)

[2.4. Biểu thức chính quy 5](#_Toc103102456)

[2.5. Quy trình vận hành 7](#_Toc103102457)

[2.6. Hạn chế 8](#_Toc103102458)

[2.7. Lỗi từ vựng 8](#_Toc103102459)

[3. TRÌNH PHÂN TÍCH CÚ PHÁP YACC 9](#_Toc103102460)

[3.1. Mô tả Yacc 9](#_Toc103102461)

[3.2. Phân tích cú pháp 9](#_Toc103102462)

[3.3. Cấu trúc của YACC 10](#_Toc103102463)

[3.4. Quy trình vận hành 12](#_Toc103102464)

[3.5. Hoạt động của bộ phân tích cú pháp 12](#_Toc103102465)

[**3.5.1.** **Phân tích cú pháp từ dưới lên (Bottom-up)** 12](#_Toc103102466)

[**3.5.2.** **Phân tích cú pháp từ trên xuống (Top-Down)** 15](#_Toc103102467)

[3.6. Xử lý lỗi cú pháp 18](#_Toc103102468)

[3.7. Chiến lược phục hồi lỗi 18](#_Toc103102469)

[3.8. Hạn chế 19](#_Toc103102470)

[4. Tham khảo 19](#_Toc103102471)

**NGHIÊN CỨU BỘ CÔNG CỤ PHÂN TÍCH TỪ VỰNG , PHÂN TÍCH CÚ PHÁP LEX VÀ YACC**

# GIỚI THIỆU

Lex và Yacc là các công cụ để sinh bộ phân tích từ vựng và phân tích cú pháp. Trước năm 1975, việc xây dựng một trình biên dịch là một quá trình tốn nhiều công sức và thời gian. Yacc được phát triển trước tiên bởi Stephen C. Johnson sau đó Lex được thiết kế ra bởi Mike E. Lesk để làm việc với Yacc. Những công cụ này đã giúp ích rất nhiều trong việc xây dựng trình biên dịch

# TRÌNH PHÂN TÍCH TỪ VỰNG LEX

* 1. **Mô tả Lex**

Lex là một công cụ xây dựng bộ phân tích từ vựng (Lexical analyzer). Nó được thiết kế để xử lý từ vựng của những dòng ký tự đầu vào. Lex chấp nhận ngôn ngữ bậc cao, đặc tả định hướng cho việc so khớp chuỗi ký tự và sinh ra chương trình đích phục vụ cho việc đoán nhận biểu thức chính quy. Biểu thức chính quy được chỉ định bởi người dùng từ nguồn đặc ta được cho bởi Lex. Lex sẽ nhận diện những biểu thức chính quy có trong chuỗi ký tự đầu vào và phân chia chúng thành các thẻ ký tự (token) tương ứng với biểu thức chính quy. Các token sau này sẽ được sử dụng bởi Yacc hoặc có thể là sản phẩm cuối cùng.

* 1. **Phân tích từ vựng**

Phân tích từ vựng (lexical analysis) là bước đầu tiên của chương trình dịch hay còn được gọi là scanning hoặc lexing Phân tích từ vựng là công việc đọc chương trình nguồn từ trái sang phải để tách ra thành các thẻ từ (token). Nói cách khác, quá trình phân tích từ vựng là quá trình dịch mà đầu nhận là các chuỗi ký tự, đầu ra là các token. Dạng đầu ra này sẽ là đầu vào của quá trình phân tích cú pháp về sau.

Ví dụ: “Tôi thích ăn cơm?”

|  |  |
| --- | --- |
| Tôi | Danh từ |
| thích | Động từ |
| ăn | Động từ |
| cơm | Danh từ |

Trong các thiết kế cổ điển: bộ phân tích từ vựng được coi như là một tiến trình song song và phụ thuộc vào bộ phân tích cú pháp, quá trình phân tích cú pháp điều khiển việc phân tích từ vựng. Bộ phân tích từ vựng sẽ đọc chương trình nguồn từ đầu hoặc từ vị trí trong lần gọi trước, tách lấy từ tố tiếp theo và trả lại cho bộ phân tích cú pháp, quá trình sẽ được lặp lại cho đến khi hết mã nguồn hoặc gặp lỗi.

Trong các thiết kế mới hơn, bộ phân tích từ vựng có xu hướng đứng tách độc lập. Điều này giúp việc thiết kế theo hướng module hóa và đơn giản hơn, giúp tăng hiệu quả hoạt động của bộ phân tích từ vựng. Giúp xử lý được ngay lập tức mỗi số lỗi cơ bản về từ vựng mà không cần phân tích cú pháp.

* 1. **Cấu trúc của LEX**

Cấu trúc của một chương trình LEX

|  |
| --- |
| %{  Phần định nghĩa  %}  %%  Phần quy tắc  %%  Các hàm hỗ trợ |

Phần định nghĩa chủ yếu là phần khai báo biến và thư viện mà ta sẽ đưa trực tiếp vào chương trình (mã C). Phần mã C được giới hạn giữa cặp ký hiệu “%{” và “%}”. Lex sẽ sao chép toàn bộ những gì giữa cặp ký hiệu “%{” và “%}” trực tiếp vào file lex.yy.c sau khi chạy Lex.

Phần thứ hai là quy tắc do người dùng định nghĩa hay các luật nhằm xác định loại token từ những ký tự nhập vào, phần quy tắc được giới hạn trong cặp ký hiệu “%%” và “%%”. Những luật này luôn bao gồm 2 phần: biểu thức chính quy và phương thức thực hiện khi nhập chuỗi phù hợp được định nghĩa trong biểu thức chính quy. Phương thức thực hiện được xác định nhờ cặp cặp ký hiệu “{” và “}”.

Phần cuối cùng là các hàm hỗ trợ cho quá trình phân tích từ vựng được viết bằng mã C và sẽ được sao chép trực tiếp vào file lex.yy.c như phần khai báo.

Ví dụ:

|  |
| --- |
| %{ // Phần định nghĩa %}  %%  [\t]+ {;}  is |  am |  are |  were |  was |  be |  being |  been |  do |  does |  did |  will |  would |  should |  can |  could |  has |  have |  had |  go | {printf("%s: is a verb\n", yytext);}  [a-zA-Z]+ {printf(“%s: is not a verb\n”, yytext)}  %%  int main() {  yylex();  return 0;  } |

* 1. **Biểu thức chính quy**

Biểu thức chính quy (Regular Expression) là một nhóm các ký tự, ký hiệu được sử dụng để mô tả những mẫu (pattern) để nhận diện chuỗi ký tự, ví dụ như nhận diện số điện thoại, địa chỉ email, số tiền, …

Bảng 2.1: Mô tả các ký tự trong biểu thức chính quy

|  |  |
| --- | --- |
| Ký tự | Ý nghĩa |
| . | Bất kỳ ký tự nào ngoài trự ký tự xuống dòng |
| \* | Lặp lại không đến nhiều lần |
| + | Lặp lại 1 hoặc nhiều lần |
| ? | Tùy chọn có hay không cho mẫu phía trước |
| ^ | Bắt đầu một dòng |
| $ | Kết thúc một dòng |
| {m, n} | Độ dài tối thiểu là m và tối đa là n |
| () | Biểu diễn một nhóm các ký tự |
| | | Biểu diễn thay thế, phép toán or |
| \ | Biểu diễn ký tự đặc biệt [ ] ( ) { } . \* + ? ^ $ \ | |
| [] | Tập hợp ký tự. Phù hợp nếu có bất kỳ ký tự nào trong dấu [] |
| [^ ] | Tập hợp ký tự phủ định. Phù hợp nếu có bất kỳ ký tự nào trong dấu [] |

Bảng 2.2: Mô tả một số biểu thức trong biểu thức chính quy

|  |  |
| --- | --- |
| Biểu thức | Ý nghĩa |
| abc | abc |
| abc\* | ab abc abcc abccc … |
| abc+ | abc abcc abccc… |
| abc)+ | abc abcbc abcbcbc … |
| a(bc)? | a abc |
| [abc] | Một trong các ký tự abc |
| [^abc] | Bất kỳ ký tự nào ngoài trừ ký tự a, b và c |
| [a-z] | Bất kỳ ký tự nào từ a đến z |
| [0-9] | Bất kỳ ký tự nào từ 0 đến 9 |
| [a-z]{3, 5} | Một trong các ký tự a đến z và có nhiều dài từ 3 đến 5 |

Bảng 2.3: Ký hiệu tắt cho tập hợp

|  |  |
| --- | --- |
| Viết tắt | Ý nghĩa |
| \w | Chữ, số và \_: [a-zA-Z0-9] |
| \W | Ngoài trừ chữ, số và \_: [^\w] |
| \d | Các số: [0-9] |
| \D | Không phải số: [^\d] |
| \s | Ký tự trắng: [\t\n\f\r\p] |
| \S | Không là ký tự trắng: [^\s] |

* 1. **Quy trình vận hành**

Việc tạo ra bộ phân tích từ vựng được thực hiện trong ba bước. Bước đầu là tạo ra chương trình lex.l bằng ngôn ngữ Lex. Sau đó chương trình llex.l sẽ được chương trình dịch của Lex xử lý và tạo ra lex.yy.c, được xây dựng từ các biểu thức chính quy có trong lex.l cùng các thủ tục chuẩn. Lex dùng bản để nhận dạng trị từ vựng. Các hành vi có liên quan đến các biểu thức chính quy trong lex.l là các chương trình con trong ngôn ngữ C và chúng được chuyển thẳng sang chương trình lex.yy.c. Cuối cùng lex.yy.c sẽ được chương trình dịch C dịch và tạo ra đối tượng a.output. Nó chính là bộ phân tích từ vựng, có khả năng chuyển các dòng nhập thành các token, được xác định bởi biểu thức chính quy đã được định nghĩa trong ngôn ngữ Lex.

**Diagram

Description automatically generated**

* 1. **Hạn chế**

Lex cũng có một vài hạn chế. Ví dụ, Lex không thể sử dụng để nhận biết các cấu trúc lồng nhau (như các dấu ngoặc đơn). Các cấu trúc lồng nhau được quản lý bởi một ngăn xếp nhất định. Khi chúng ta gặp một dấu “(“ chúng ta đẩy nó vào ngăn xếp, còn khi tìm thấy dấu “)” chúng ta so khớp dó với các phần tử ở đỉnh ngăn xếp và lấy phần tử đó ra khỏi ngăn xếp. Lex không có ngăn xếp nên nó không phù hợp để phân tích cấu trúc lồng nhau.

* 1. **Lỗi từ vựng**

Chỉ một số ít lỗi được phát hiện tại bước phân tích từ vựng, bởi vì bộ phân tích từ vựng có nhiều cách nhìn nhận chương trình nguồn. Ví dụ chuỗi “**fi**” được nhìn thấy lần đầu trong một chương trình C với ngữ cảnh; **fi (a == f(x)) …** Bộ phân tích từ vựng không thể biết đây là lỗi không được viết đúng từ khóa “if” hay một danh biểu chưa được khai báo. Vì “**fi**” là một danh biểu hợp lệ nên bộ phân tích từ vựng phải trả về một token và để một giai đoạn khác sau xác định lỗi. Tuy nhiên, trong một vài tình huống phải khắc phục lỗi để phân tích tiếp. Chiến lược đơn giản nhất là “phương thức hoảng sợ” (panic mode): Các ký tự tiếp theo sẽ được xóa ra khỏi chuỗi nhập còn lại cho đến khi tìm ra một token hoàn chỉnh. Kỹ thuật này đôi khi cũng gây ra sự nhầm lẫn cho giai đoạn phân tích cú pháp, nhưng nói chung là vẫn có thể sử dụng được.

# TRÌNH PHÂN TÍCH CÚ PHÁP YACC

* 1. **Mô tả Yacc**

Yacc là một công cụ xây dựng bộ phân tích cú pháp (Parser). Một trình phân tích cú pháp hoàn chỉnh thường gồm có hai phần: một lexer , còn được gọi là máy quét hoặc tokenizer , và phân tích cú pháp thích hợp. Trình phân tích cú pháp cần lexer vì nó không hoạt động trực tiếp trên văn bản mà trên đầu ra được tạo bởi lexer.

* 1. **Phân tích cú pháp**

Phân tích cú pháp (Syntax Analysis hay Parsing) là giai đoạn thứ hai của trình biên dịch. Bộ phân tích cú pháp nhận chuỗi các token từ bộ phân tích từ vựng và xác nhận rằng chuỗi này có thể được sinh ra từ văn phạm của ngôn ngữ nguồn bằng cách tạo ra cây phân tích cú pháp cho chuỗi.

Diagram

Description automatically generatedVí dụ: với chuỗi nhập a = b + c \* 60, ta có cây phân tích cú pháp:

* 1. **Cấu trúc của YACC**

Tương tự như LEX, cấu trúc của chương trình YACC gồm 3 phần:

|  |
| --- |
| %{  Phần định nghĩa  %}  %%  Phần luật dịch  %%  Các hàm hỗ trợ |

Phần định nghĩa bao gồm 2 phần nhỏ. Một là phần khai bao C đặt nằm trong cặp dấu “%{” và “%}”. Hai là khai báo các token. Nằm ngoài cặp dấu “%{” và “%}”.

Phần luật dịch chứa các luật dịch mà người lập trình định nghĩa. Mỗi luật dịch là một văn phạm kết hợp với hành vi ngữ nghĩa.

Mỗi văn phạm có dạng:

<vế trái> -> <alt1> | <alt2> | <alt3> | … | <altn>

Khi văn phạm được mô tả trong Yacc:

<vế trái>: <alt1> {hành vi ngữ nghĩa 1}

| <alt2> {hành vi ngữ nghĩa 2}

| <alt2> {hành vi ngữ nghĩa 2}

…

| <altn> {hành vi ngữ nghĩa n}

;

Cuối cùng là các hàn hỗ trợ được viết bằng ngôn ngữ C

Để minh họa cho chương trình nguồn của Yacc, dưới đây là code minh hoạt một chương trình máy tính bỏ túi đơn giản, đọc một biểu thức số học, ước lượng ròi in ra giá trị số của nó.

|  |
| --- |
| %{  #include <stdio.h>  #include <stdlib.h>  extern int yylex();  void yyerror(char \*msg);  %}  %union { float f; }  %token <f> NUM  %type <f> E T F  %%  S : E {printf("%f\n", $1);}  ;    E : E '+' T {$$ = $1 + $3;}  | E '-' T {$$ = $1 - $3;}  | T {$$ = $1;}  ;  T : T '\*' F {$$ = $1 \* $3;}  | T '/' F {$$ = $1 / $3;}  | F {$$ = $1;}  ;  F : '(' E ')' {$$ = $2;}  | '-' F {$$ = -$2;}  | NUM {$$ = $1;}  ;  %%  void yyerror(char \*msg){  fprintf(stderr, "%s\n", msg);  exit(1);  }  int main(){  yyparse();  return 0;  } |

* 1. Diagram

     Description automatically generated**Quy trình vận hành**

Một chương trình dịch có thể được xây dựng nhờ Yacc bằng phương thức được minh họa trong hình trên. Trước tiên, cần chuẩn bị một tập tin, chẳng hạn là translate.y, chứa một đặc tả Yacc của chương trình dịch. Lệnh yacc translate.y biến đổi tập tin translate.y thành một chương trình C có tên là y.tab.C . Bằng cách dịch y.tab.C cùng với thư viện ly nhờ lệnh cc y.tab.C - ly chúng ta thu được một chương trình đối tượng a.out thực hiện quá trình dịch được đặc tả bởi chương trình Yacc ban đầu. Nếu cần thêm các thủ tục khác, chúng có thể được biên dịch hoặc được tải vào y.tab.C giống như mọi chương trình C khác

* 1. **Hoạt động của bộ phân tích cú pháp**
     1. **Phân tích cú pháp từ dưới lên (Bottom-up)**

Diagram

Description automatically generatedPhân tích cú pháp từ dưới lên bắt đầu từ các nút là của cây và hoạt động theo hướng đi lên cho đến khi nó đến nút gốc.

* + - 1. **Trình phân tích cú pháp Shift – Reduce**

Phân tích cú pháp Shift – Reduce xây dựng một cây phân tích cú pháp cho chuỗi nhập bắt đầu từ nút lá và đi lên hướng về nút gốc. Đây có thể xem là quá trình thu gọn (reduce) một chuỗi đầu vào thành một kỳ hiệu bắt đầu của văn phạm. Tại mỗi bước thu gọn, một chuỗi con cụ thể đối sánh được với vế phải của một luật sinh nào đó thì chuỗi con này sẽ được thay thế bởi ký hiệu vế trái của luật sinh đó.

Handle của một chuỗi là một chuỗi con hợp với vế phải của luật sinh và nếu chúng ta thu gọn nó thành vế trái của luật sinh đó thì có thể dẫn đến ký hiệu chưa kết thúc bắt đầu

Ví dụ: Xét văn phạm

E -> E + E

E -> E \* E

E -> (E)

E -> id

Chuỗi dẫn xuất phải:

E => E + E (các handle được gạch dưới)

* E + E \* E
* E + E \* id3
* E + id2 \* id3
* id1 + id2 \* id3

Cấu tạo của trình phân tích cú pháp Shift-Reduce

* Dùng một Stack để lưu các ký hiệu văn phạm
* Dùng một bộ đệm nhập INPUT đẻ giữ chuỗi nhập cần phân tích
* Ta dùng ký hiệu “$” để đánh dấu đáy Stack và xác định cuối chuỗi

Hoạt động của trình phân tích cú pháp Shift-Reduce

* Khởi đầu thì Stack rỗng và w nằm trong bộ đệm input.
* Bộ phân tích đẩy các ký hiệu vào trong Stack cho đến khi một handle nằm trên đỉnh Stack
* Thu gọn handle thành một vế trái của một luật sinh nào đó.
* Lặp lại bước 2 và 3 cho đến khi gặp một lỗi hoặc Stack chứa ký hiệu bắt đầu và bộ đệm Input rỗng (thông báo kết thúc thành công).

|  |  |  |
| --- | --- | --- |
| STACK | INPUT | ACTION |
| $id1 | id1 + id2 \* id3 $ | Shift |
| $ E | + id2 \* id3 $ | Reduce E -> id |
| $ E + | + id2 \* id3 $ | Shift |
| $ E + id2 | id2 \* id3 $ | Shift |
| $ E + E | \* id3 $ | Reduce E -> id |
| $ E + E \* | \* id3 $ | Shift |
| $ E + E \* id3 | id3 $ | Shift |
| $ E + E \* E | $ | Reduce E -> id |
| $ E + E | $ | Reduce E -> E \* E |
| $ E | $ | Reduce E -> E + E |
|  |  | Chấp nhận |

* + - 1. **Trình phân tích cú pháp LR**

Một phương pháp tổng quát hơn của kỹ thuật Shift - Reduce là phân tích cú pháp LR (LR parsing: left–to–right, right most derivation)

* A shift-reduce parser cố gắng thu gọn chuỗi nhập về ký hiệu S.
* Tại mỗi bước rút gọn, chuỗi con của dòng ký hiệu nhập so khớp với vế phải của luật sinh nào đó sẽ thay thế bởi ký hiệu chưa kết thúc bên vế trái của luật sinh đó.
* Nếu xâu con được chọn đúng, thì dẫn xuất phải nhất của chuỗi nhập sẽ được tạo ra theo thứ tự ngược lại.

Trình phân tích cú pháp LR còn được gọi là LR(k), có thể sử dụng để phân tích một lớp rộng các văn phạm phi ngữ cảnh

* L(Left-to-right): Duyệt chuỗi nhập từ trái sang phải
* R(Rightmost derivation): Xây dựng chuỗi dẫn xuất phải nhất đảo ngược
* k: Số lượng ký hiệu lookahead tại mỗi thời điểm, dùng để đưa ra quyết định phân tích. Khi không đề cập đến k, chúng ta hiểu ngầm là k = 1

Ưu điểm của trình phân tích cú pháp LR:

* Có thể nhận biết hầu như tất cả các ngôn ngữ lập trình được tạo ra bởi văn phạm phi ngữ cảnh
* Phương pháp phân tích cú pháp LR là phương pháp tổng quát của phương pháp shift-reduce không quay lui
* Lớp văn phạm có thể dùng phương pháp LR là một lớp rộng lớn hơn lớp văn phạm có thể sử dụng phương pháp dự đoán
* Có thể xác định lỗi cú pháp nhanh ngay trong khi duyệt dòng nhập từ trái sang phải

Nhược điểm của phân tích cú pháp LR: Xây dựng LR parser khá phức tạp

* + - 1. **Các trình phân tích cú pháp còn lại**

**SLR** – Trình phân tich cú pháp LR đơn giản

* Hoạt động trên lớp ngữ pháp nhỏ
* Ít trạng thái
* Triển khai đơn giản, nhanh chóng

**LR** – Trình phân tích cú pháp LR

* Làm việc trên bộ ngữ pháp hoàn chỉnh
* Nhiều trạng thái
* Triển khai chậm

**LALR** – Trình phân tích cú pháp LR nhìn trước

* Hoạt động trên bộ ngữ pháp trung bình
* Số lượng trạng thái giống như SLR
  + 1. **Phân tích cú pháp từ trên xuống (Top-Down)**

Phân tích cú pháp từ trên xuống có thể được xem như một nỗ lực tìm kiếm một dẫn xuất trái nhất cho chuỗi nhập. Nó cũng có thể xem như một nỗ lực xây dựng cây phân tích cú pháp từ nút gốc và phát sinh dần xuống lá.

Ví dụ: Cho các văn phạm:

S -> aABc

A -> bc

B -> d

Diagram

Description automatically generated Chuỗi đầu vào: “abcde”

**Diagram, schematic

Description automatically generated**Các trình phân tích cú pháp từ trên xuống:

* + - 1. **Đệ quy đi xuống (Recursive Descent)**

Đệ quy đi xuống (Recursive Descent): Một dạng tổng quát của kỹ thuật phân tích từ trên xuống, gọi là phân tích cú pháp đệ quy lùi, có thể quay lui để quét lại chuỗi nhập. Tuy nhiên, dạng này thường rất ít gặp. Lý do là với các kết cấu ngôn ngữ lập trình, chúng ta hiếm khi dùng đến nó.

* + - 1. **Quay lui (Back-tracking)**

Quay lui (Back-tracking): nếu một dẫn xuất của một sản phẩm không thành công, trình phân tích cú pháp sẽ khởi động lại quy trình bằng cách sử dụng các quy tắc khác nhau của cùng một sản phẩm. Kỹ thuật này có thể xử lý chuỗi đầu vào nhiều lần để xác định sản phẩm phù hợp.

Ví cho: Cho văn phạm:

S -> aAb | aBb

A -> cx | dx

B -> xe

Chuỗi đầu vào: “adxb”

A picture containing triangle

Description automatically generated

Đối với một chuỗi đầu vào, trình phân tích hoạt động như sau:

* Bắt đầu bằng S từ các quy tắc với chữ cái ngoài cùng bên trái của chuỗi đầu vào, tức là ‘a’. Nhiều quy tắc của S phù hợp với nó và chọn từ trái (S -> aAb)
* Tiếp theo trình phân tích tiến tới ký tự tiếp theo là ‘d’
* Trình phân tích mở rộng A và kiểm tra quy tắccó nó từ bên trái (A -> cx). Nó không hợp với kí hiệu đầu vào
* Vậy trình phân tích quay lui để thu được quy tắc tiếp theo của A (A -> dx)
* Bây giờ trình phân tích cú pháp khớp với tất cả các ký tự chuỗi đầu vào. Chuỗi được chấp nhận
  + - 1. **Phân tích cú pháp dự đoán (Predictive Parser)**

Trình phân tích cú pháp dự đoán (Predictive Parser) là trình phân tích cú pháp gốc đệ quy, có khả năng dự đoán sản phẩm nào sẽ được sử dụng để thay thế chuỗi đầu vào. Trong nhiều trường hợp, bằng cách viết văn phạm một cách cẩn thận, loại bỏ đệ qui trái ra khỏi văn phạm rồi tạo ra yếu tố trái, chúng ta có thể thu được một văn phạm mà một bộ phân tích cú pháp đệ quy lùi phân tích được, nhưng không cần quay lui, gọi là phân tích cú pháp dự đoán.

* + - 1. **Phân tích cú pháp LL (LL Parser)**

Trình phân tích cú pháp LL chấp nhận ngữ pháp LL (Left to right, Left most derivation). Ngữ pháp LL là một tập hợp con ngữ pháp không có ngữ pháp nhưng với một số hạn chế để có được phiên bản đơn giản hóa, nhằm dễ dàng triển khai.

* 1. **Xử lý lỗi cú pháp**

Chương trình nguồn có thể chứa các lỗi ở nhiều mức độ khác nhau:

* Lỗi từ vựng: như danh biểu, từ khóa, toán tử viết không đúng
* Lỗi cú pháp: như ghi một biểu thức toán học với cấu dấu ngoặc đóng mở không cân bằng
* Lỗi ngữ nghĩa: như một cặp toán tử áp dụng vào một toán hàng không tương thích
* Lỗi logic: như thực hiện một lời gọi đệ qui không thể kết thúc

Phần lớn việc phát hiện và phục hồi lỗi trong một trình biên dịch tập trung vào giai đoạn phân tích cú pháp. Vì thế, bộ xử lý lỗi trong quá trình phân tích cú pháp phải đặt mục đích sau:

* Ghi nhận và thông báo lỗi một cách rõ ràng và chính xác
* Phục hồi lỗi một cách nhanh chóng để có thể xác định các lỗi tiếp theo
* Không làm chậm tiến trình của một tiến chương trình đúng
  1. **Chiến lược phục hồi lỗi**

Phục hồi lỗi là kỹ thuật vượt qua lỗi để tiếp tục quá trình dịch. Nhiều chiến lược phục hồi lỗi có thể dùng trong bộ phân tích cú pháp. Mặc dù không có chiến lược nào được chấp nhận hoàn toàn, nhưng một số trong chúng đã được áp dụng rộng rãi. Ở đây, chúng ta giới thiệu một số chiến lược:

* **Phương thức “hoảng sợ” (panic mode recovery):** Đây là phương pháp đơn giản nhất cho cài đặt và có thể dùng cho hầu hết các phương pháp phân tích. Khi một lỗi được phát hiện thì bộ phân tích cú pháp bỏ qua từng ký hiệu một cho đến khi tìm thấy một tập hợp được chỉ định của các token đồng bộ (synchronizing tokens), các token đồng bộ thường là dấu chấm phẩy.
* **Chiến lược mức ngữ đoạn(phrase-level recovery):** khi phát hiện một lỗi, bộ phân tích cú pháp có thể thực hiện sự hiệu chỉnh cục bộ trên phần còn lại của dòng nhập. Cụ thể là thay thế phần đầu còn lại bằng một chuỗi ký tự có thể tiếp tục. Chẳng hạn, dấu phẩy, bởi dấu chấm phẩy, xóa một dấu phẩy lạ hoặc thêm vào một dấu chấm phẩy.
* **Chiến lược dùng các luật sinh sửa lỗi (error production):** Thêm vào văn phạm của ngôn ngữ những luật sinh lỗi và sử dụng văn phạm này để xây dựng bộ phân tích cú pháp, chúng ta có thể sinh ra bộ đoán lỗi thích hợp để chỉ ra cấu trúc lỗi được nhận biết trong dòng nhập
  1. **Hạn chế**

Yacc không thể phân tích văn phạm nhập nhằng, nghĩa là cùng một đầu vào có thể khớp với 2 cây cú pháp trở lên

Yacc cũng không thể phân tích văn phạm cần nhiều hơn một token nhìn trước để biết nó khớp với luật nào.

# Tham khảo

[1] http://dinosaur.compilertools.net/

[2] http://flex.sourceforge.net

[3] http://gnuwin32.sourceforge.net/packages/bison.html

[4] https://github.com/ckshitij/LEX-AND-YACC

[5] <https://comdy.vn/trinh-bien-dich/phan-tich-cu-phap/>

[6] https://web.iitd.ac.in/~sumeet/flex\_\_bison.pdf