# CHƯƠNG IV
# PHÂN TÍCH CÚ PHÁP

**Nội dung chính:**
Mỗi ngôn ngữ lập trình đều có các quy tắc diễn tả cấu trúc cú pháp của các chương trình có định dạng đúng. Các cấu trúc cú pháp này được mô tả bởi *văn phạm phi ngữ cảnh*. Phần đầu của chương nhắc lại khái niệm văn phạm phi ngữ cảnh, cách tìm một văn phạm tương đương không còn đệ quy trái và mơ hồ. Phần lớn nội dung của chương trình bày các phương pháp phân tích cú pháp thường được sử dụng trong các trình biên dịch: *Phân tích cú pháp từ trên xuống* (Top down) và *Phân tích cú pháp từ dưới lên* (Bottom up). Các chương trình nguồn có thể chứa các lỗi cú pháp. Trong quá trình phân tích cú pháp chương trình nguồn, sẽ rất bất tiện nếu chương trình dừng và thông báo lỗi khi gặp lỗi đầu tiên. Vì thế cần phải có kỹ thuật để vượt qua các lỗi cú pháp để tiếp tục quá trình dịch - Các kỹ thuật *phục hồi lỗi*. Từ văn phạm đặc tả ngôn ngữ lập trình và lựa chọn phương pháp phân tích cú pháp phù hợp, sinh viên có thể tự mình xây dựng một *bộ phân tích cú pháp*. Phần còn lại của chương giới thiệu công cụ *Yacc*. Sinh viên có thể sử dụng công cụ này để tạo bộ phân tích cú pháp thay vì phải tự cài đặt. Mô tả chi tiết về Yacc được tìm thấy ở phần phụ lục B.

**Mục tiêu cần đạt:**
Sau khi học xong chương này, sinh viên phải nắm được:

- Các phương pháp phân tích cú pháp và các chiến lược phục hồi lỗi.
- Cách tự cài đặt một bộ phân tích cú pháp từ một văn phạm phi ngữ cảnh xác định.
- Cách sử dụng công cụ Yacc để sinh ra bộ phân tích cú pháp.

**Kiến thức cơ bản:**
Sinh viên phải có các kiến thức về:

- Văn phạm phi ngữ cảnh (Context Free Grammar – CFG), Automat đẩy xuống (Pushdown Automata – PDA).
- Cách biến đổi từ một CFG về một PDA.

**Tài liệu tham khảo:**

[1] **Automata and Formal Language. An Introduction** – Dean Kelley – Prentice Hall, Englewood Cliffs, New Jersey 07632.

[2] **Compilers : Principles, Technique and Tools** - Alfred V.Aho, Jeffrey D.Ullman - Addison - Wesley Publishing Company, 1986.

[3] **Compiler Design** – Reinhard Wilhelm, Dieter Maurer - Addison - Wesley Publishing Company, 1996.

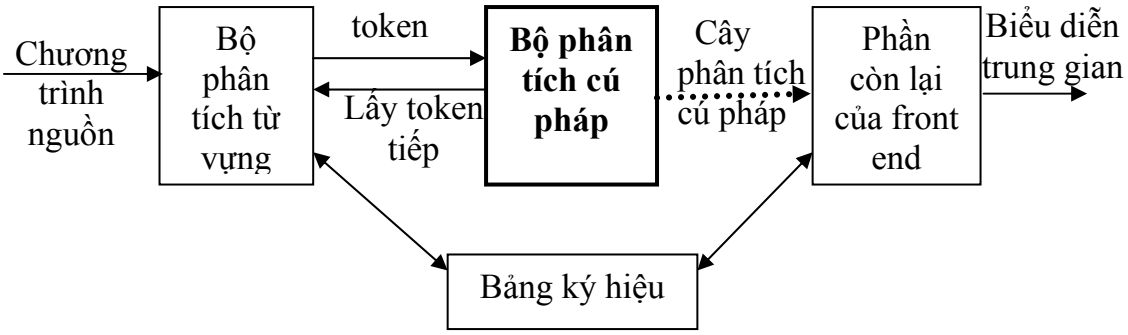
[4] **Design of Compilers : Techniques of Programming Language Translation** - Karen A. Lemone - CRC Press, Inc, 1992.

[5] **Modern Compiler Implementation in C -** Andrew W. Appel - Cambridge University Press, 1997.

# I. VAI TRÒ CỦA BỘ PHÂN TÍCH CÚ PHÁP

## 1. Vai trò của bộ phân tích cú pháp

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. Bộ phân tích cú pháp cũng có cơ chế ghi nhận các lỗi cú pháp theo một phương thức linh hoạt và có khả năng phục hồi được các lỗi thường gặp để có thể tiếp tục xử lý phần còn lại của chuỗi nhập.




**Hình 4.1 -** *Vị trí của bộ phân tích cú pháp trong mô hình trình biên dịch*

## 2. 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 các dấu ngoặc đóng và mở không cân bằng.

**- Lỗi ngữ nghĩa** như một toán tử áp dụng vào một toán hạng 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 biện dịch tập trung vào giai đoạn phân tích cú pháp. Vì thế, bộ xử lý lỗi (error handler) trong quá trình phân tích cú pháp phải đạt 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 chương trình đúng.



## 3. Các chiến lược phục hồi lỗi

Phục hồi lỗi là kỹ thuật vượt qua các 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 :

**a. 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 (;) hoặc end.

**b. 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.

**c. 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.

**d. Chiến lược hiệu chỉnh toàn cục (global correction):** Một cách lý tưởng là trình biên dịch tạo ra một số thay đổi trong khi xử lý một lỗi. Có những giải thuật để lựa chọn một số tối thiểu các thay đổi để đạt được một hiệu chỉnh có chi phí toàn cục nhỏ nhất. Cho một chuỗi nhập có lỗi x và một văn phạm G, các giải thuật này sẽ tìm được một cây phân tích cú pháp cho chuỗi y mà số lượng các thao tác chèn, xóa và thay đổi token cần thiết để chuyển x thành y là nhỏ nhất. Nói chung, hiện nay kỹ thuật này vẫn còn ở dạng nghiên cứu lý thuyết.

## II. BIẾN ĐỔI VĂN PHẠM PHI NGỮ CẢNH



Nhiều ngôn ngữ lập trình có cấu trúc đệ quy mà nó có thể được định nghĩa bằng các văn phạm phi ngữ cảnh (context-free grammar) G với 4 thành phần G (V, T, P, S), trong đó:

- V : là tập hữu hạn các **ký hiệu chưa kết thúc** hay các biến (variables)
- T : là tập hữu hạn các **ký hiệu kết thúc** (terminals).
- P : là tập **luật sinh** của văn phạm (productions).
- S ∈ V: là **ký hiệu bắt đầu** của văn phạm (start symbol).

**Ví dụ 4.1:** Văn phạm với các luật sinh sau cho phép định nghĩa các biểu thức số học đơn giản (với E là một biểu thức expression) :

$$E \rightarrow E\ A\ E \mid (E) \mid -E \mid \mathbf{id}$$

$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$



### 1. Cây phân tích cú pháp và dẫn xuất

Cây phân tích cú pháp có thể được xem như một dạng biểu diễn hình ảnh của một dẫn xuất. Ta nói rằng $\alpha A\beta$ dẫn xuất ra $\alpha\gamma\beta$ (ký hiệu: $\alpha A\beta \Rightarrow \alpha\gamma\beta$) nếu $A \rightarrow \gamma$ là một luật sinh, $\alpha$ và $\beta$ là các chuỗi tùy ý các ký hiệu văn phạm.

Nếu $\alpha_1 \Rightarrow \alpha_2 \Rightarrow .. .. \Rightarrow \alpha_n$ ta nói $\alpha_1$ dẫn xuất ra (suy ra) $\alpha_n$

**Ký hiệu** $\Rightarrow$ : dẫn xuất ra qua 1 bước

$\Rightarrow^*$ : dẫn xuất ra qua 0 hoặc nhiều bước.

$\Rightarrow^+$ : dẫn xuất ra qua 1 hoặc nhiều bước.

Ta có tính chất:

1. $\alpha \Rightarrow^* \alpha$ với $\forall\alpha$
2. $\alpha \Rightarrow^* \beta$ và $\beta \Rightarrow^* \gamma$ thì $\alpha \Rightarrow^* \gamma$

Cho một văn phạm G với ký hiệu bắt đầu S. Ta dùng quan hệ $\Rightarrow^+$ để định nghĩa L(G) một ngôn ngữ được sinh ra bởi G. Chuỗi trong L(G) có thể chỉ chứa một ký hiệu kết thúc của G. Chuỗi các ký hiệu kết thúc w thuộc L(G) nếu và chỉ nếu $S \Rightarrow^+ w$, chuỗi w được gọi là một câu của G. Một ngôn ngữ được sinh ra bởi một văn phạm gọi là ngôn ngữ phi ngữ cảnh. Nếu hai văn phạm cùng sinh ra cùng một ngôn ngữ thì chúng được gọi là hai văn phạm tương đương.

Nếu $S \Rightarrow^* \alpha$, trong đó $\alpha$ có thể chứa một ký hiệu chưa kết thúc thì ta nói rằng $\alpha$ là một dạng câu (sentential form) của G. Một câu là một dạng câu có chứa toàn các ký hiệu kết thúc.

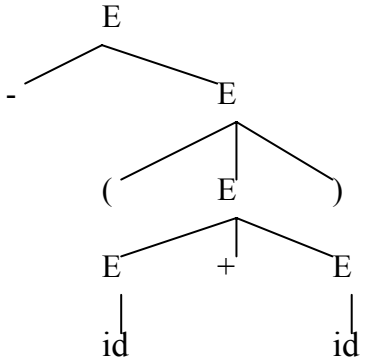
Một cây phân tích cú pháp có thể xem như một biểu diễn đồ thị cho một dẫn xuất.

Để hiểu được bộ phân tích cú pháp làm việc ta cần xét dẫn xuất trong đó chỉ có ký hiệu chưa kết thúc trái nhất trong bất kỳ dạng câu nào được thay thế tại mỗi bước, dẫn xuất như vậy được gọi là trái nhất. Nếu $\alpha \Rightarrow \beta$ trong đó ký hiệu chưa kết thúc trái nhất trong $\alpha$ được thay thế, ta viết $\alpha \Rightarrow^*_{lm} \beta$

Nếu $S \Rightarrow^*_{lm} \alpha$ ta nói $\alpha$ là dạng câu trái của văn phạm.

Tương tự, ta có dẫn xuất phải nhất - còn gọi là dẫn xuất chính tắc (canonical derivations)

**Ví dụ 4.2:** Cây phân tích cú pháp cho chuỗi nhập : **- (id + id)** sinh từ văn phạm trong ví dụ 4.1



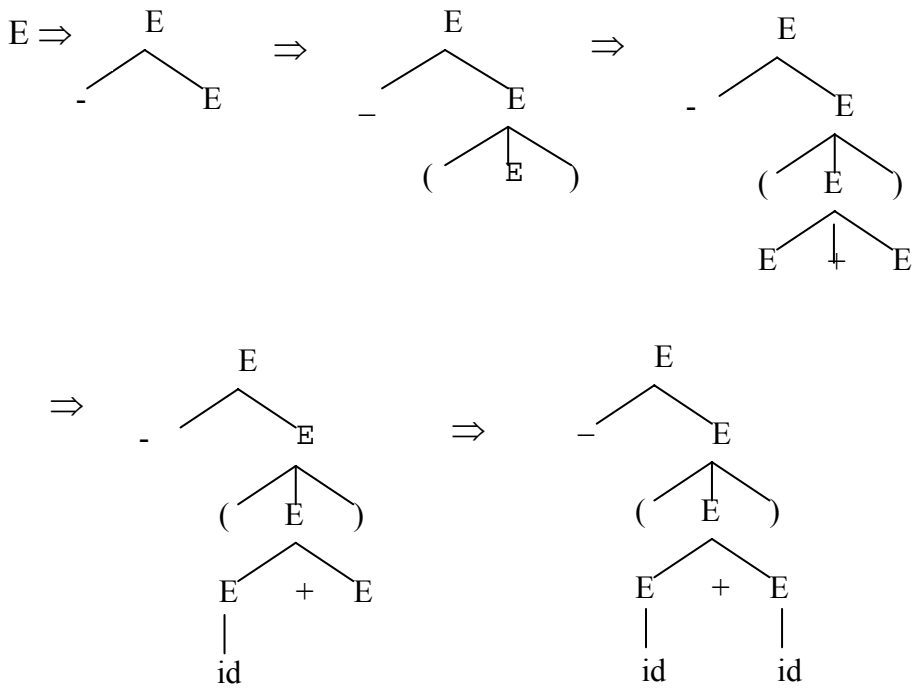

**Hình 4.2** - *Minh họa một cây phân tích cú pháp*

Để thấy mối quan hệ giữa cây phân tích cú pháp và dẫn xuất, ta xét một dẫn xuất : $\alpha_1 \Rightarrow \alpha_2 \Rightarrow .. .. \Rightarrow \alpha_n$ trong đó $\alpha_i$ là một ký hiệu chưa kết thúc A.

Với mỗi $\alpha_i$ ta xây dựng một cây phân tích cú pháp. Ví dụ với dẫn xuất:

$E \Rightarrow -E \Rightarrow - (E) \Rightarrow - (E + E) \Rightarrow - (id + E) \Rightarrow - (id + id)$

Ta có quá trình xây dựng cây phân tích cú pháp như sau :




**Hình 4.3 -** *Xây dựng cây phân tích cú pháp từ dẫn xuất*



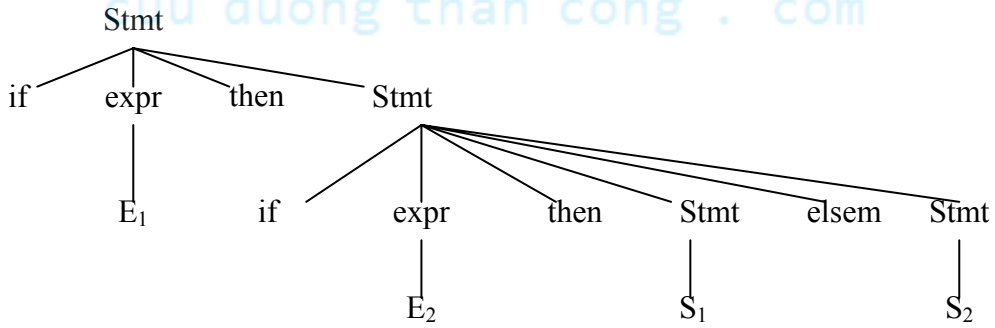
## 2. Loại bỏ sự mơ hồ

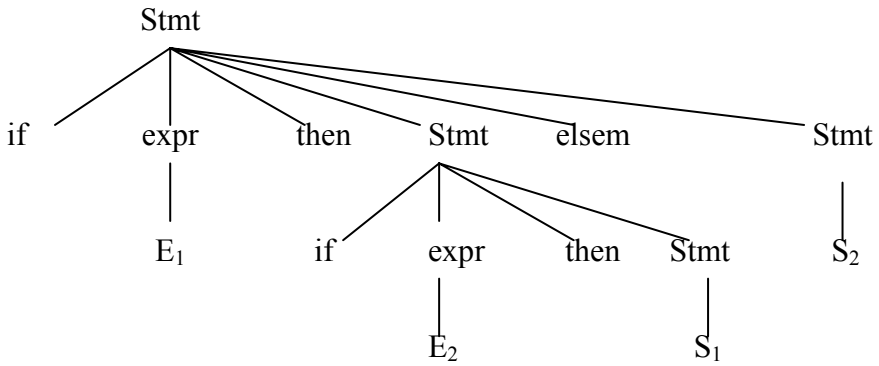
Một văn phạm tạo ra nhiều hơn một cây phân tích cú pháp cho cùng một chuỗi nhập được gọi là văn phạm mơ hồ. Nếu một văn phạm là mơ hồ, ta không thể xác định được cây phân tích cú pháp nào sẽ được chọn. Vì thế, ta phải viết lại một văn phạm nhằm tránh sự mơ hồ của nó. Một ví dụ, chúng ta sẽ loại bỏ sự mơ hồ trong văn phạm sau :

Stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| **other**

Đây là một văn phạm mơ hồ vì câu nhập **if E1 then if E2 then S1 else S2** sẽ có hai cây phân tích cú pháp :








**Hình 4.4 -** *Hai cây phân tích cú pháp cho một câu nhập*

Để tránh sự mơ hồ này ta đưa ra nguyên tắc "Khớp mỗi else với một then chưa khớp gần nhất trước đó". Với qui tắc này, ta viết lại văn phạm trên như sau :

Stmt → matched_stmt | unmatched_stmt

matched_stmt → **if** expr **then** matched_stmt **else** matched_stmt
| **other**

unmatched_stmt → **if** expr **then** Stmt
| **if** expr **then** matched_stmt **else** unmatched_stmt

Văn phạm tương đương này sinh ra tập chuỗi giống như văn phạm mơ hồ trên, nhưng nó chỉ có một cách dẫn xuất ra cây phân tích cú pháp cho từng chuỗi nhập.



### 3. Loại bỏ đệ qui trái

Một văn phạm là đệ qui trái (left recursive) nếu nó có một ký hiệu chưa kết thúc A sao cho có một dẫn xuất $A \Rightarrow^{+} A\alpha$, với $\alpha$ là một chuỗi nào đó. Các phương pháp phân tích từ trên xuống không thể nào xử lý văn phạm đệ qui trái, do đó cần phải dùng một cơ chế biến đổi tương đương để loại bỏ các đệ qui trái.

Đệ qui trái có hai loại :

**Loại trực tiếp**: Dạng $A \rightarrow A\alpha$

**Loại gián tiếp**: $A \Rightarrow^{i} A\alpha$ với $i \geq 2$

Xét văn phạm như sau:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$



Biến S cũng là biến đệ qui trái vì $S \Rightarrow Aa \Rightarrow Sda$, nhưng đây không phải là đệ qui trái trực tiếp.

**. Với đệ qui trái trực tiếp**: Luật sinh có dạng:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

Sẽ thay thế bởi :

$$\begin{cases} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \varepsilon \end{cases}$$

. **Với đệ qui trái gián tiếp** (và nói chung là đệ qui trái, ta sử dụng giải thuật sau)

- **Giải thuật 4.1**: **Loại bỏ đệ qui trái**

**Input:** Văn phạm không tuần hoàn và không có các luật sinh ε (nghĩa là văn phạm không chứa các dạng $A \Rightarrow^{+} A$ và $A \rightarrow \varepsilon$)

**Output:** Văn phạm tương đương không đệ qui trái

**Phương pháp:**

1. Sắp xếp các ký hiệu không kết thúc theo thứ tự $A_1, A_2, ..., A_n$
2. **For** i:=1 **to** n **do**

**Begin**

**for** j:=1 **to** i -1 **do**

**begin**

Thay luật sinh dạng $A_i \rightarrow A_j\gamma$ bởi luật sinh $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$

trong đó $A_j \rightarrow \delta_1 \mid \delta_2 \mid ... \mid \delta_k$ là tất cả các $A_i$ luật sinh hiện tại;

**end;**

Loại bỏ đệ qui trái trực tiếp trong số các $A_i$ luật sinh;

**End;**

**Ví dụ 4.3**: Áp dụng thuật toán trên cho văn phạm ví dụ trên. Về lý thuyết, thuật toán 4.1 không bảo đảm sẽ hoạt động được trong trường hợp văn phạm có chứa các luật sinh ε, nhưng trong trường hợp này luật sinh $A \rightarrow \varepsilon$ rõ ràng là "vô hại".

1. Sắp xếp các ký hiệu chưa kết thúc theo thứ tự S, A.
2. Với i = 1, không có đệ qui trái trực tiếp nên không có điều gì xảy ra.

Với i = 2, thay các S - luật sinh vào $A \rightarrow Sd$ được: $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Loại bỏ đệ qui trái trực tiếp cho các A luật sinh, ta được :

$S \rightarrow Aa \mid b$

$A \rightarrow bdA'$

$A' \rightarrow cA' \mid adA \mid \varepsilon$

## 4. Tạo ra yếu tố trái

Tạo ra yếu tố trái (left factoring) là một phép biến đổi văn phạm rất có ích để có được một văn phạm thuận tiện cho việc phân tích dự đoán. Ý tưởng cơ bản là khi không rõ luật sinh nào trong hai luật sinh khả triển có thể dùng để khai triển một ký hiệu chưa kết thúc A, chúng ta có thể viết lại các A - luật sinh nhằm "hoãn" lại việc quyết định cho đến khi thấy đủ nguyên liệu cho một lựa chọn đúng.

Xét văn phạm cho câu lệnh if:

stmt → **if** expr **then** stmt **else** stmt

| **if** expr **then** stmt

Khi gặp token if, chúng ta không thể quyết định ngay cần chọn luật sinh nào để triển khai cho stmt. Để giải quyết vấn đề này, một cách tổng quát, khi có luật sinh dạng $A \to \alpha\beta_1 \mid \alpha\beta_2$, ta biến đổi luật sinh thành dạng :

$$\begin{cases} A \to \alpha A' \\ A' \to \beta_1 \mid \beta_2 \end{cases}$$

- **Giải thuật 4.2 : Tạo yếu tố trái cho văn phạm**

**Input:** Văn phạm G

**Output:** Văn phạm tương đương với yếu tố trái.

**Phương pháp:**

Với mỗi ký hiệu chưa kết thúc A, có các ký hiệu dẫn đầu các vế phải giống nhau, ta tìm một chuỗi $\alpha$ là chuỗi có độ dài lớn nhất chung cho tất cả các vế phải ($\alpha$ là yếu tố trái). Giả sử $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$, trong đó $\gamma$ không có chuỗi dẫn đầu chung với các vế phải khác. Ta biến đổi luật sinh thành :

$$\begin{cases} A \to \alpha A' \mid \gamma \\ A' \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n \end{cases}$$

Với A' là ký hiệu chưa kết thúc mới. Áp dụng lặp đi lặp lại phép biến đổi này cho đến khi không còn hai khả triển nào cho một ký hiệu chưa kết thúc có một tiền tố chung.



**Ví dụ 4.4**: Áp dụng thuật toán 4.2 cho văn phạm sau:

$S \to$ **i** E **t** S | **i** E **t** S eS | $\alpha$

$E \to b$

Ta có văn phạm tương đương có chứa yếu tố trái như sau :

$S \to$ **i** E **t** S S' | $\alpha$

$S' \to$ eS | $\varepsilon$

$E \to$ b

## III. PHÂN TÍCH CÚ PHÁP TỪ TRÊN XUỐNG

Trong mục này, chúng ta giới thiệu các ý niệm cơ bản về phương pháp **phân tích cú pháp từ trên xuống** (Top Down Parsing) và trình bày một dạng không quay lui hiệu quả của phương pháp phân tích từ trên xuống, gọi là phương pháp **phân tích dự đoán** (predictive parser). Chúng ta định nghĩa một lớp văn phạm LL(1) (viết tắt của Left-to-right parse, Leftmost-derivation, 1-symbol lockahead ), trong đó phân tích dự đoán có thể xây dựng một cách tự động.

### 1. Phân tích cú pháp đệ qui lùi (Recursive Descent Parsing)

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 bắt đầu từ nút gốc và phát sinh dần xuống lá. 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.

## 2. Bộ phân tích cú pháp dự đoán (Predictive Parser)

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.

**Xây dựng sơ đồ dịch cho bộ phân tích dự đoán:**

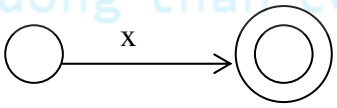
Để xây dựng sơ đồ dịch cho phương pháp phân tích xuống, trước hết loại bỏ đệ qui trái, tạo yếu tố trái cho văn phạm. Sau đó thực hiện các bước sau cho mỗi ký hiệu chưa kết thúc A :

1. Tạo một trạng thái khởi đầu và một trạng thái kết thúc.

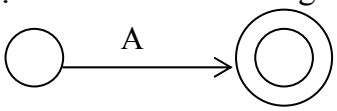
2. Với mỗi luật sinh $A \rightarrow X_1X_2 \ldots X_n$ , tạo một đường đi từ trạng thái khởi đầu đến trạng thái kết thúc bằng các cạnh có nhãn $X_1X_2 \ldots X_n$

Một cách cụ thể, **sơ đồ dịch được vẽ theo các nguyên tắc** sau:

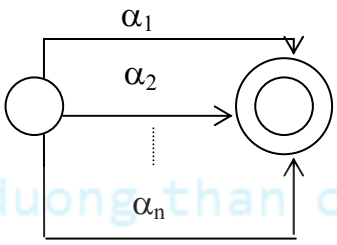
1. Mỗi ký hiệu chưa kết thúc tương ứng với một sơ đồ dịch trong đó nhãn cho các cạnh là token hoặc ký hiệu chưa kết thúc.
2. Mỗi token tương ứng với việc đoán nhận token đó và đọc token kế tiếp



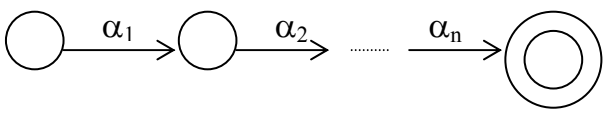

3. Mỗi ký hiệu chưa kết thúc tương ứng với lời gọi thủ tục cho ký hiệu đó.




4. Mỗi luật sinh có dạng $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ tương ứng với sơ đồ dịch




5. Mỗi luật sinh dạng $A \rightarrow \alpha_1 \ \alpha_2 .. \ .. \ \alpha_n$ tương ứng với sơ đồ dịch




**Ví dụ 4.5**: Xét văn phạm sinh biểu thức toán học

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$F \rightarrow (E) \mid \mathbf{id}$

Loại bỏ đệ quy trái trong văn phạm, ta được văn phạm tương đương sau :

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

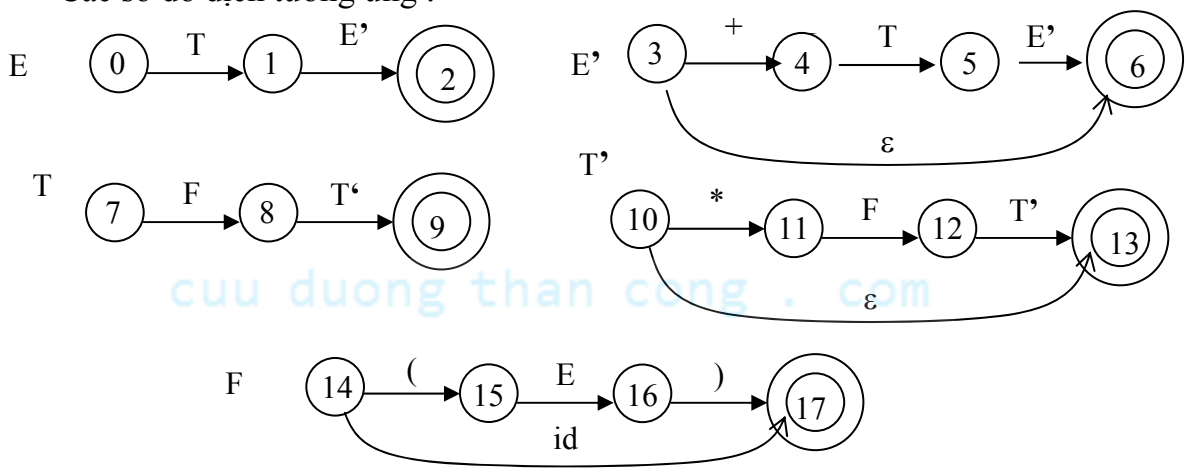
$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

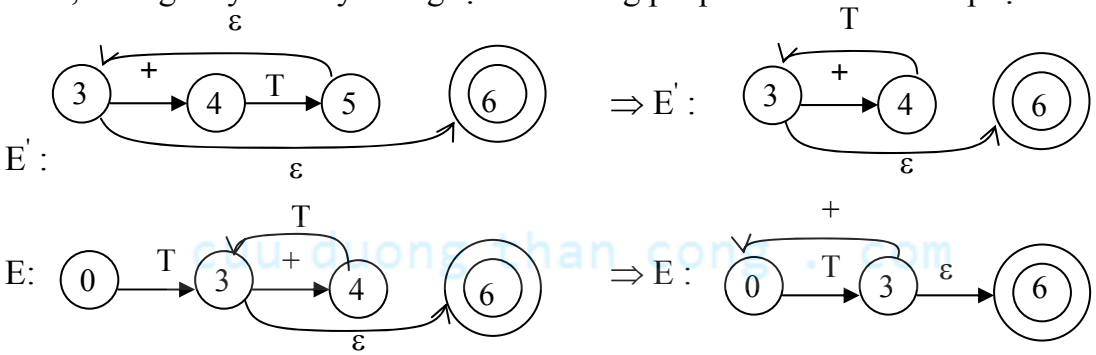
Một chương trình phân tích cú pháp dự đoán được thiết kế dựa trên sơ đồ dịch cho các ký hiệu chưa kết thúc trong văn phạm. Nó sẽ cố gắng so sánh các ký hiệu kết thúc với chuỗi nguyên liệu và đưa ra lời gọi đệ qui mỗi khi nó phải đi theo một cạnh có nhãn là ký hiệu chưa kết thúc.

Các sơ đồ dịch tương ứng :

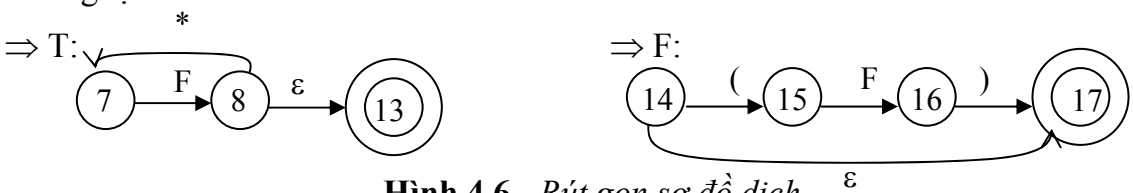



**Hình 4.5 -** *Các sơ đồ dịch cho các ký hiệu văn phạm*

Các sơ đồ dịch có thể được đơn giản hóa bằng cách thay sơ đồ này vào sơ đồ khác, những thay thế này tương tự như những phép biến đổi trên văn phạm.



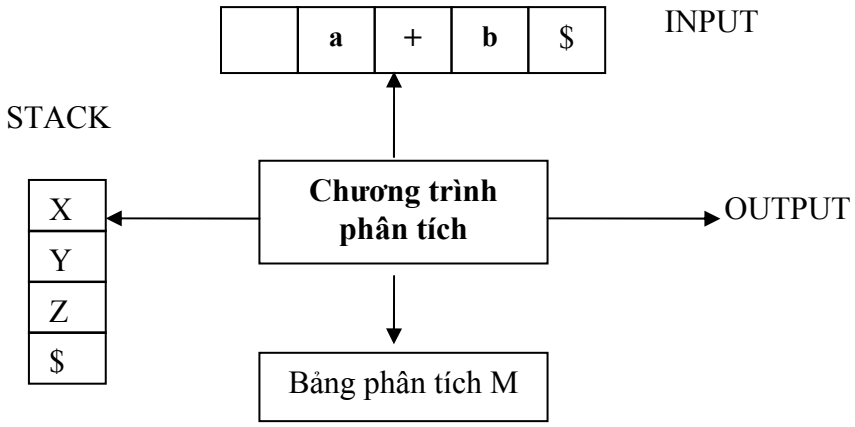

Tương tự ta có:




**Hình 4.6 -** *Rút gọn sơ đồ dịch*

**Phân tích dự đoán không đệ qui**

Chúng ta có thể xây dựng bộ phân tích dự đoán không đệ qui bằng cách duy trì tường minh một Stack chứ không phải ngầm định qua các lời gọi đệ quy. Vấn đề chính trong quá trình phân tích dự đoán là việc xác định luật sinh sẽ được áp dụng cho một biến ở bước tiếp theo. Một bộ phân tích dự đoán sẽ làm việc theo mô hình sau:




**Hình 4.7** - *Mô hình bộ phân tích cú pháp dự đoán không đệ quy*

**- INPUT** là bộ đệm chứa chuỗi cần phân tích, kết thúc bởi ký hiệu $.

**- STACK** chứa một chuỗi các ký hiệu văn phạm với ký hiệu $ nằm ở đáy Stack.

**- Bảng phân tích M** là một mảng hai chiều dạng M[A,a], trong đó A là ký hiệu chưa kết thúc, a là ký hiệu kết thúc hoặc $.

Bộ phân tích cú pháp được điều khiển bởi chương trình hoạt động như sau: Chương trình xét ký hiệu X trên đỉnh Stack và ký hiệu nhập hiện hành a. Hai ký hiệu này xác định hoạt động của bộ phân tích cú pháp như sau:

1. Nếu X = a = $ thì chương trình phân tích cú pháp kết thúc thành công.

2. Nếu X = a $\neq$ $, Pop X ra khỏi Stack và đọc ký hiệu nhập tiếp theo.

3. Nếu X là ký hiệu chưa kết thúc thì chương trình truy xuất đến phần tử M[X,a] trong bảng phân tích M:

   - Nếu M[X,a] là một luật sinh có dạng X $\rightarrow$ UVW thì Pop X ra khỏi đỉnh Stack và Push W, V, U vào Stack (với U trên đỉnh Stack), đồng thời bộ xuất sinh ra luật sinh X $\rightarrow$ UVW.
   - Nếu M[X,a] = error, gọi chương trình phục hồi lỗi.

- **Giải thuật 4.3 : Phân tích cú pháp dự đoán không đệ quy.**

  **Input:** Chuỗi nhập w và bảng phân tích cú pháp M cho văn phạm G.

  **Output:** Nếu w $\in$ L (G), cho ra một dẫn xuất trái của w. Ngược lại, thông báo lỗi.

  **Phương pháp:**

Khởi đầu Stack chứa ký hiệu chưa kết thúc bắt đầu (S) trên đỉnh và bộ đệm chứa câu nhập dạng w$.

Đặt con trỏ ip trỏ tới ký hiệu đầu tiên của w$ ;

***Repeat***

Gọi X là ký hiệu trên đỉnh Stack và a là ký hiệu được trỏ bởi ip ;

**If** X là ký hiệu kết thúc hoặc $ **then**

**If** X = a **then** lấy X ra khỏi Stack và dịch chuyển ip

**else** error ( )

**Else** // X là ký hiệu chưa kết thúc

**If** M[X,a] = $X \rightarrow Y_1\ Y_2 \ldots Y_k$ **then**

**begin**

Lấy X ra khỏi Stack;

Đẩy $Y_k, Y_{k-1}, \ldots, Y_1$ vào Stack;

Xuất ra luật sinh $X \rightarrow Y_1\ Y_2 \ldots Y_k$;

**end**

**else** error ( ) /* Stack rỗng */

***Until*** X = $

**Ví dụ 4.6:** Xét văn phạm đã được khử đệ qui trái sinh biểu thức toán học trong ví dụ 4.5 :

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$



Bảng phân tích M của văn phạm được cho như sau : (ô trống tương ứng với lỗi)

| **Ký hiệu chưa kết thúc** | **Ký hiệu nhập** | | | | | |
|---|---|---|---|---|---|---|
| | **id** | **+** | ***** | **(** | **)** | **$** |
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | $E \rightarrow +TE'$ | | | $E \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| **T'** | | $T' \rightarrow \varepsilon$ | $T' \rightarrow *FT'$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |
| **F** | $F \rightarrow id$ | | | $F \rightarrow (E)$ | | |



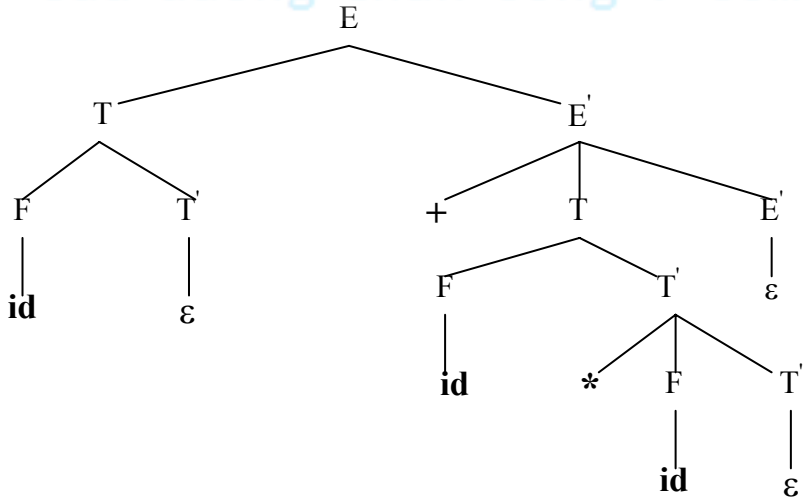
**Hình 4.8 -** *Bảng phân tích cú pháp M cho văn phạm*

Quá trình phân tích cú pháp cho chuỗi nhập: **id + id * id** được trình bày trong bảng sau :

| **STACK** | **INPUT** | **OUTPUT** |
|---|---|---|
| $ E | id + id * id $ | |
| $ E' T | id + id * id $ | $E \rightarrow T\ E'$ |

| | | |
|---|---|---|
| $ E' T' F | id + id * id $ | T → F T' |
| $ E' T' id | id + id * id $ | F → id |
| $ E' T' | + id * id $ | |
| $ E' | + id * id $ | T' → ε |
| $ E' T + | + id * id $ | E' → + T E' |
| $ E' T | id * id $ | |
| $ E' T' F | id * id $ | T → F T' |
| $ E' T' id | id * id $ | F → id |
| $ E' T' | * id $ | |
| $ E' T' F * | * id $ | T' → * F T' |
| $ E' T' F | id $ | F → id |
| $ E' T' id | id $ | |
| $ E' T' | $ | T' → ε |
| $ E' | $ | E' → ε |
| $ | $ | |

Cây phân tích cú pháp được hình thành từ output :






**Nhận xét:**

- Mỗi văn phạm có một bảng phân tích M tương ứng.
- Chương trình không cần thay đổi cho các văn phạm khác nhau.

### 3. Hàm FIRST và FOLLOW

**FIRST** và **FOLLOW** là các tập hợp cho phép xây dựng bảng phân tích M và phục hồi lỗi theo chiến lược panic_mode.

- **Định nghĩa FIRST(α)**: Giả sử α là một chuỗi các ký hiệu văn phạm, FIRST(α) là tập hợp các ký hiệu kết thúc mà nó bắt đầu một chuỗi dẫn xuất từ α.

Nếu $\alpha \Rightarrow^* \varepsilon$ thì $\varepsilon \in FIRST(\alpha)$.

**Cách tính FIRST(X)**: Thực hiện các quy luật sau cho đến khi không còn có ký hiệu kết thúc nào hoặc ε có thể thêm vào tập FIRST(X) :

1. Nếu X là kí hiệu kết thúc thì FIRST(X) là {X}
2. Nếu $X \rightarrow \varepsilon$ là một luật sinh thì thêm ε vào FIRST(X).
3. Nếu $X \rightarrow Y_1Y_2Y_3 ...Y_k$ là một luật sinh thì thêm tất cả các ký hiệu kết thúc khác ε của $FIRST(Y_1)$ vào FIRST(X). Nếu $\varepsilon \in FIRST(Y_1)$ thì tiếp tục thêm vào FIRST(X) tất cả các ký hiệu kết thúc khác ε của $FIRST(Y_2)$. Nếu $\varepsilon \in FIRST(Y_1) \cap FIRST(Y_2)$ thì thêm tất cả các ký hiệu kết thúc khác $\varepsilon \in FIRST(Y_3)$ ... Cuối cùng thêm ε vào FIRST(X) nếu $\varepsilon \in \cap^k_{i=1} FIRST(Y_i)$

**Ví dụ 4.7**: Với văn phạm sau:

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Theo định nghĩa tập FIRST, ta có :

Vì $F \Rightarrow (E) \mid id \Rightarrow FIRST(F) = \{ (, id \}$

Từ $T \rightarrow F\ T'$ và $\varepsilon \notin FIRST(F) \Rightarrow FIRST(T) = FIRST(F)$

Từ $E \rightarrow T\ E'$ và $\varepsilon \notin FIRST(T) \Rightarrow FIRST(E) = FIRST(T)$

Vì $E' \rightarrow \varepsilon \Rightarrow \varepsilon \in FIRST(E')$

Mặt khác do $E' \rightarrow +\ TE'$ mà $FIRST(+) = \{+\} \Rightarrow FIRST(E') = \{+, \varepsilon \}$

Tương tự $FIRST(T') = \{*, \varepsilon \}$

Vậy ta có :

$FIRST(E) = FIRST(T) = FIRST(F) = \{ (, id \}$

$FIRST(E') = \{+, \varepsilon \}$

$FIRST(T') = \{*, \varepsilon \}$



- **Định nghĩa FOLLOW(A)**: (với A là một ký hiệu chưa kết thúc) là tập hợp các ký hiệu kết thúc a mà nó xuất hiện ngay sau A (bên phía phải của A) trong một dạng câu nào đó. Tức là tập hợp các ký hiệu kết thúc a, sao cho tồn tại một dẫn xuất dạng $S \Rightarrow^* \alpha A a \beta$. Chú ý rằng nếu A là ký hiệu phải nhất trong một dạng câu nào đó thì $\$ \in FOLLOW(A)$ ($ là ký hiệu kết thúc chuỗi nhập ).

**Cách tính FOLLOW(A)**: Áp dụng các quy tắc sau cho đến khi không thể thêm gì vào mọi tập FOLLOW được nữa.

1. Đặt $ vào follow(S), trong đó S là ký hiệu bắt đầu của văn phạm và $ là ký hiệu kết thúc chuỗi nhập.

2. Nếu có một luật sinh A→ αBβ thì thêm mọi phần tử khác ε của FIRST(β)vào trong FOLLOW(B).
3. Nếu có luật sinh A→ αB hoặc A→ αBβ mà ε ∈ FIRST(β) thì thêm tất cả các phần tử trong FOLLOW(A) vào FOLLOW(B).

**Ví dụ 4.8:** Với văn phạm trong ví dụ 4.6 nói trên:

Áp dụng luật 2 cho luật sinh F→ (E) ⇒ **)** ∈ FOLLOW(E) ⇒ FOLLOW(E) ={$, ) }

Áp dụng luật 3 cho E → TE' ⇒ ), $ ∈ FOLLOW(E') ⇒ FOLLOW(E') ={$, ) }

Áp dụng luật 2 cho E → TE' ⇒ + ∈ FOLLOW(T).

Áp dụng luật 3 cho E' → +TE' , E' → ε

⇒ FOLLOW(E') ⊂ FOLLOW(T) ⇒ FOLLOW(T) = { +, ), $ }.

Áp dụng luật 3 cho T→ FT' thì FOLLOW(T') = FOLLOW(T) ={+, ), $ }

Áp dụng luật 2 cho T→ FT' ⇒ * ∈ FOLLOW(F)

Áp dụng luật 3 cho T' → * F T' , T'→ ε

⇒ FOLLOW(T') ⊂ FOLLOW(F) ⇒ FOLLOW(F) = {*, +, ), $ }.

Vậy ta có: FOLLOW(E) = FOLLOW(E') = { $, ) }

FOLLOW(T) = FOLLOW(T') = { +, ), $ }

FOLLOW(F) = {*,+, ), $ }



## 4. Xây dựng bảng phân tích M

- **Giải thuật 4.4 : Xây dựng bảng phân tích cú pháp dự đoán**

**Input:** Văn phạm G.

**Output:** Bảng phân tích cú pháp M.

**Phương pháp:**

1. Với mỗi luật sinh A→ α của văn phạm, thực hiện hai bước 2 và 3.
2. Với mỗi ký hiệu kết thúc a ∈ FIRST(α), thêm A→ α vào M[A,a].
3. Nếu ε ∈ FIRST(α) thì đưa luật sinh A→ α vào M[A,b] với mỗi ký hiệu kết thúc b ∈ FOLLOW(A). Nếu ε ∈ FIRST(α) và $ ∈ FOLLOW(A) thì đưa luật sinh A→ α vào M[A,$].
4. Ô còn trống trong bảng tương ứng với lỗi (error).



**Ví dụ 4.9:** Áp dụng thuật toán trên cho văn phạm trong ví dụ 4.6. Ta thấy:

Luật sinh E → TE' : Tính FIRST(TE') = FIRST(T) = {(,id}

⇒ M[E,id] và M[E,( ] chứa luật sinh E → TE'

Luật sinh E'→ + TE' : Tính FIRST(+TE') = FIRST(+) = {+}

⇒ M[E',+] chứa E' → +TE'

Luật sinh E' → ε : Vì ε ∈ FIRST(E') và FOLLOW(E') = { ), $ }

⇒ E → ε nằm trong M[E',)] và M[E',$]

Luật sinh T'→ * FT' : FIRST(* FT') = {* }

⇒ T' → * FT' nằm trong M[T',*]

Luật sinh T' → ε: Vì ε ∈ FIRST(T') và FOLLOW(T') = {+, ), $}

⇒ T' → ε nằm trong M[T', +] , M[T', )] và M[T',$]

Luật sinh F→ (E) ; FIRST((E)) = { ( }

⇒ F → ( E) nằm trong M[F, (]

Luật sinh F → id ; FIRST(id) = {id}

⇒ F → id nằm trong M[F, id]

Bảng phân tích cú pháp M của văn phạm được xây dựng như trong hình 4.8.

## 5. Văn phạm LL(1)

Giải thuật 4.4 có thể áp dụng cho bất kỳ văn phạm G nào để sinh ra bảng phân tích M. Tuy nhiên, có những văn phạm (đệ quy trái hoặc mơ hồ) thì trong bảng phân tích M sẽ có thể có những ô đa trị (có chứa nhiều hơn 1 luật sinh).

**Ví dụ 4.10:** Xét văn phạm sau:



S → **i**E **t** S S' | a

S' → **e**S | ε

E → b

Bảng phân tích cú pháp M của văn phạm như sau :

| **Ký hiệu chưa kết thúc** | **Ký hiệu kết thúc** | | | | | |
|---|---|---|---|---|---|---|
| | **a** | **b** | **e** | **i** | **T** | **$** |
| **S** | S→ a | | | S→ iEtSS' | | |
| **S'** | | | S → ε<br>S' → eS | | | S'→ε |
| **E** | | E→b | | | | |



**Hình 4.9 -** *Bảng phân tích cú pháp M cho văn phạm ví dụ 4.10*

Đây là một văn phạm mơ hồ và sự mơ hồ này được thể hiện qua việc chọn luật sinh khi gặp ký hiệu **e** (else). Ô tại vị trí M [S', e] được gọi là ô đa trị.

Một văn phạm mà bảng phân tích M không có các` ô đa trị được gọi là **văn phạm LL(1)** với ý nghĩa như sau :

**L:** Left-to-right parse (mô tả hành động quét chuỗi nhập từ trái sang phải)

**L:** Leftmost-derivation (biểu thị việc sinh ra một dẫn xuất trái cho chuỗi nhập)

**1:** 1-symbol lookahead (tại mỗi một bước, đầu đọc chỉ đọc trước được một token để thực hiện các quyết định phân tích cú pháp)

Văn phạm LL(1) có một số tính chất đặc biệt. Không có văn phạm mơ hồ hay đệ quy trái nào có thể là LL(1). Người ta đã chứng minh rằng một văn phạm G là LL(1) nếu và chỉ nếu mỗi khi $A \rightarrow \alpha \mid \beta$ là 2 luật sinh phân biệt của G, các điều kiện sau đây sẽ đúng:

1. Không có một ký hiệu kết thúc a nào mà cả $\alpha$ và $\beta$ đều dẫn xuất ra các chuỗi bắt đầu bằng a.

2. Tối đa chỉ có $\alpha$ hoặc chỉ có $\beta$ có thể dẫn xuất ra chuỗi rỗng.

3. Nếu $\beta \Rightarrow^* \varepsilon$ thì $\alpha$ không dẫn xuất được chuỗi nào bắt đầu bằng một ký hiệu kết thúc thuộc tập FOLLOW(A).

Rõ ràng văn phạm trong ví dụ 4.5 cho các biểu thức số học là LL(1), nhưng văn phạm trong ví dụ 4.10 là văn phạm mô hình hóa câu lệnh **if - then - else** không phải là LL(1).

Vấn đề đặt ra bây giờ là làm thế nào để giải quyết các ô đa trị? Một phương án khả thi là biến đổi văn phạm bằng cách loại bỏ mọi đệ quy trái, rồi tạo yếu tố trái khi có thể được với mong muốn sẽ sinh ra một văn phạm với bảng phân tích cú pháp không chứa ô đa trị nào. Nhưng cũng có một số văn phạm mà không có cách gì biến đổi thành văn phạm LL(1). Nói chung, không có quy tắc tổng quát nào để biến một ô đa trị thành ô đơn trị mà không làm ảnh hưởng đến ngôn ngữ đang được nhận dạng bởi bộ phân tích cú pháp.



Khó khăn chính khi dùng một bộ phân tích cú pháp dự đoán là việc viết một văn phạm cho ngôn ngữ nguồn. Việc loại bỏ đệ quy trái và tạo yếu tố trái tuy dễ thực hiện nhưng chúng biến đổi văn phạm trở thành khó đọc và khó dùng cho các mục đích biên dịch.

## 6. Phục hồi lỗi trong phân tích dự đoán

Một lỗi sẽ được tìm thấy trong quá trình phân tích dự đoán khi:

1. Ký hiệu kết thúc trên đỉnh Stack không phù hợp với token kế tiếp trong dòng nhập. Hoặc :

2. Trên đỉnh Stack là ký hiệu chưa kết thúc A, token trong dòng nhập là a nhưng M[A,a] rỗng.



Phục hồi lỗi theo phương pháp **panic_mode** là bỏ qua các ký hiệu trong dòng nhập cho đến khi gặp một phần tử trong **tập hợp các token đồng bộ** (synchronizing token).

Tính hiệu quả của phương pháp này tùy thuộc vào cách chọn tập hợp các token đồng bộ. Một số heuristics có thể là:

1. Ta có thể đưa tất cả các ký hiệu trong FOLLOW(A) vào trong tập hợp token đồng bộ cho ký hiệu chưa kết thúc A.
2. FOLLOW(A) cũng chưa phải là một tập hợp các token đồng bộ cho A. Ví dụ, các lệnh của C kết thúc bởi **;** (dấu chấm phẩy ). Nếu một lệnh thiếu dấu ; thì từ khóa của lệnh kế tiếp sẽ bị bỏ qua. Thông thường ngôn ngữ có cấu trúc

phân cấp, ví dụ biểu thức nằm trong một lệnh, lệnh nằm trong một khối lệnh, v.v. Chúng ta có thể thêm vào tập hợp đồng bộ của một cấu trúc những ký hiệu mà nó bắt đầu cho một cấu trúc cao hơn. Ví dụ, ta có thể thêm các từ khoá bắt đầu cho các lệnh vào tập đồng bộ cho ký hiệu chưa kết thúc sinh ra biểu thức.

3. Nếu chúng ta thêm các phần tử của FIRST(A) vào tập đồng bộ cho ký hiệu chưa kết thúc A thì quá trình phân tích có thể hòa hợp với A nếu một ký hiệu trong FIRST(A) xuất hiện trong dòng nhập.

**Ví dụ 4.11:** Sử dụng các ký hiệu kết thúc trong tập FOLLOW làm token đồng bộ hóa hoạt động khá hữu hiệu khi phân tích cú pháp cho các biểu thức trong văn phạm ví dụ 4.6.

FOLLOW(E) = FOLLOW(E') = { $, )}

FOLLOW(T) = FOLLOW(T') = { +,$, )}

FOLLOW(F) = {*,+, $, )}

Bảng phân tích M cho văn phạm này được thêm vào các ký hiệu đồng bộ "**synch**", lấy từ tập FOLLOW của các ký hiệu chưa kết thúc - xác định các token đồng bộ :

| **Ký hiệu chưa kết thúc** | **Ký hiệu kết thúc** | | | | | |
|---|---|---|---|---|---|---|
| | **id** | **+** | ***** | **(** | **)** | **$** |
| **E** | E→TE' | | | E→TE' | **synch** | **synch** |
| **E'** | | E'→ +TE' | | | E'→ ε | E'→ ε |
| **T** | T→ FT' | **synch** | | T→ FT' | **synch** | **synch** |
| **T'** | | T'→ ε | T'→ *FT' | | T'→ ε | T'→ ε |
| **F** | F→ id | **synch** | **synch** | F→ (E) | **synch** | **synch** |

**Hình 4.10 -** *Bảng phân tích cú pháp M phục hồi lỗi*

Bảng này được sử dụng như sau:

- ✓ Nếu M[A,a] là rỗng thì bỏ qua token a.
- ✓ Nếu M[A,a] là "**synch**" thì lấy A ra khỏi Stack nhằm tái hoạt động quá trình phân tích.
- ✓ Nếu một token trên đỉnh Stack không phù hợp với token trong dòng nhập thì lấy token ra khỏi Stack.

Chẳng hạn, với chuỗi nhập : **+ id * + id**, bộ phân tích cú pháp và cơ chế phục hồi lỗi thực hiện như sau :

| **STACK** | **INPUT** | **OUTPUT** |
|---|---|---|
| $ E | + id * + id $ | **error, nhảy qua +** |
| $ E | id * + id $ | E → T E' |
| $ E' T | id * + id $ | T → F T' |

| | | |
|---|---|---|
| $ E' T' F | id * + id $ | F → id |
| $ E' T' id | id * + id $ | |
| $ E' T' | * + id $ | T' → * F T' |
| $ E' T' F * | * + id $ | |
| $ E' T' F | + id $ | **error**, M[F,+] = **synch** pop F |
| $ E' T' | + id $ | T → ε |
| $ E' | + id $ | E' → + T E' |
| $ E' T + | + id $ | |
| $ E' T | id $ | T' → F T' |
| $ E' T' F | id $ | F→ id |
| $ E' T' id | id $ | T' → ε |
| $ E' T' | $ | E' → ε |
| $ E' | $ | |
| $ | $ | |

## IV. PHÂN TÍCH CÚ PHÁP TỪ DƯỚI LÊN

Phần này sẽ giới thiệu một kiểu phân tích cú pháp từ dưới lên tổng quát gọi là **phân tích cú pháp Shift -Reduce**. Một dạng dễ cài đặt của nó gọi là **phân tích cú pháp thứ bậc toán tử** (Operator - Precedence parsing) cũng sẽ được trình bày và cuối cùng, một phương pháp tổng quát hơn của kỹ thuật Shift - Reduce là **phân tích cú pháp LR** (LR parsing) sẽ được thảo luận.

### 1. Bộ phân tích cú pháp Shift - Reduce

Phân tích cú pháp Shift - Reduce cố gắng 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 w thành một ký 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. Và nếu chuỗi con được chọn đúng tại mỗi bước, một dẫn xuất phải đảo ngược sẽ được xây dựng.

**Ví dụ 4.12:** Cho văn phạm:

S → **a** A B **e**

A→ A **b c** | **b**

B → **d**

Câu abbcde có thể thu gọn thành S theo các bước sau:

a b b c d e

a A b c d e

a A d e

a A B e

S

Thực chất đây là một dẫn xuất phải nhất đảo ngược như sau :

$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$

*(Dẫn xuất phải nhất là chuỗi các thay thế ký hiệu chưa kết thúc phải nhất)*

## 2. Handle

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 4.13:** Xét văn phạm sau:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow$ **id**

Chuỗi dẫn xuất phải :

$E \Rightarrow_{rm} E + E$ (các handle được gạch dưới)

$\Rightarrow_{rm} E + \underline{E * E}$

$\Rightarrow_{rm} E + E * \underline{\mathbf{id}_3}$

$\Rightarrow_{rm} E + \underline{\mathbf{id}_2} * \underline{\mathbf{id}_3}$

$\Rightarrow_{rm} \underline{\mathbf{id}_1} + \underline{\mathbf{id}_2} * \underline{\mathbf{id}_3}$

## 3. Cắt tỉa handle (Handle Pruning)

Handle pruning là kỹ thuật dùng để tạo ra dẫn xuất phải nhất đảo ngược từ chuỗi ký hiệu kết thúc w mà chúng ta muốn phân tích.

Nếu w là một câu của văn phạm thì $w = \gamma_n$. Trong đó, $\gamma_n$ là dạng câu phải thứ n của dẫn xuất phải nhất mà chúng ta chưa biết.

$S \Rightarrow \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \ldots\ldots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$

Để xây dựng dẫn xuất này theo thứ tự ngược lại, chúng ta tìm handle $\beta_n$ trong $\gamma_n$ và thay thế $\beta_n$ bởi An ($A_n$ là vế trái của luật sinh $A_n \rightarrow \beta_n$) để được dạng câu phải thứ n -1 là $\gamma_{n-1}$. Quy luật trên cứ tiếp tục. Nếu ta có một dạng câu phải $\gamma_0 = S$ thì sự phân tích thành công.

**Ví dụ 4.14**: Với văn phạm:

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

Và câu nhập: **id1 + id2 * id3** , ta có các bước thu gọn câu nhập thành ký hiệu bắt đầu E như sau :

| Dạng câu phải | Handle | Luật thu gọn |
|---|---|---|
| $id_1 + id_2 * id_3$ | $id_1$ | $E \rightarrow id$ |
| $E + id_2 * id_3$ | $id_2$ | $E \rightarrow id$ |
| $E + E * id_3$ | $id_3$ | $E \rightarrow id$ |
| $E + E * E$ | $E * E$ | $E \rightarrow E * E$ |
| $E + E$ | $E + E$ | $E \rightarrow E + E$ |
| $E$ | | Thành công |

## 4. Cài đặt bộ phân tích cú pháp Shift - Reduce

Có hai vấn đề cần phải giải quyết nếu chúng ta dùng kỹ thuật phân tích cú pháp này. Thứ nhất là định vị chuỗi con cần thu gọn trong dạng câu dẫn phải, và thứ hai là xác định luật sinh nào sẽ được dùng nếu có nhiều luật sinh chứa chuỗi con đó ở vế phải.

**Cấu tạo:**

Dùng 1 Stack để lưu các ký hiệu văn phạm.

Dùng 1 bộ đệm nhập INPUT để giữ chuỗi nhập cần phân tích w.

Ta dùng ký hiệu $ để đánh dấu đáy Stack và xác định cuối chuỗi nhập.



**Hoạt động:**

1. Khởi đầu thì Stack rỗng và w nằm trong bộ đệm input.

2. Bộ phân tích đẩy các ký hiệu nhập vào trong Stack cho đến khi một handle β nằm trên đỉnh Stack.

3. Thu gọn β thành vế trái của một luật sinh nào đó.

4. 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).

**Ví dụ 4.15:** Với văn phạm $E \rightarrow E + E \mid E * E \mid (E) \mid$ **id** và câu nhập **id1 + id2 * id3**

Quá trình phân tích cú pháp sẽ thực hiện như sau:

| STACK | INPUT | ACTION |
|---|---|---|
| $ | $id_1 + id_2 * id_3$ $ | Đẩy |
| $ $id_1$ | $+ id_2 * id_3$ $ | Thu gọn bởi $E \rightarrow id$ |
| $ E | $+ id_2 * id_3$ $ | Đẩy |
| $ E + | $id_2 * id_3$ $ | Đẩy |
| $ E + $id_2$ | $* id_3$ $ | Thu gọn bởi $E \rightarrow id$ |
| $ E + E | $* id_3$ $ | Đẩy |
| $ E + E * | $id_3$ $ | Đẩy |
| $ E + E * $id_3$ | $ | |

| | | |
|---|---|---|
| $ E + E * E | $ | Thu gọn bởi E → id |
| $ E + E | $ | Thu gọn bởi E → E * E |
| $ E | $ | Thu gọn bởi E → E + E |
| | | Chấp nhận |

## V. PHÂN TÍCH CÚ PHÁP THỨ BẬC TOÁN TỬ

Lớp văn phạm có tính chất không có luật sinh nào có vế phải là ε hoặc có hai ký hiệu chưa kết thúc nào nằm kế nhau có thể dễ dàng xây dựng bộ phân tích cú pháp Shift- Reduce hiệu quả theo lối thủ công. Một trong những kỹ thuật phân tích dễ cài đặt nhất gọi là phân tích cú pháp thứ bậc toán tử.

### 1. Quan hệ thứ tự ưu tiên

Bảng định nghĩa 3 quan hệ thứ bậc được cho như sau :

| Quan hệ | Ý nghĩa |
|---|---|
| $a <\bullet b$ | a có độ ưu tiên thấp hơn b |
| $a \doteq b$ | a có độ ưu tiên bằng b |
| $a \bullet> b$ | a có độ ưu tiên cao hơn b |

**Hình 4.11 -** *Các quan hệ thứ bậc toán tử*

### 2. Sử dụng quan hệ thứ tự ưu tiên của toán tử

Các quan hệ ưu tiên này giúp việc xác định handle.

Trước hết, ta dựa vào các quy tắc sau để xây dựng bảng quan hệ ưu tiên giữa các ký hiệu kết thúc.

1. Nếu toán tử $\theta_1$ có độ ưu tiên cao hơn $\theta_2$ thì $\theta_1 \bullet> \theta_2$ và $\theta_2 <\bullet \theta_1$;

   (E + E * E + E thì handle là E * E).

2. Nếu toán tử $\theta_1$ có độ ưu tiên bằng $\theta_2$ thì :

   . $\theta_1 \bullet> \theta_2$ và $\theta_2 \bullet> \theta_1$ nếu các toán tử là kết hợp trái.

   . $\theta_1 <\bullet \theta_2$ và $\theta_2 <\bullet \theta_1$ nếu các toán tử là kết hợp phải.

**Ví dụ 4.16:** Toán tử + và - có độ ưu tiên bằng nhau và kết hợp trái nên:

$+ \bullet> +$ ; $+ \bullet> -$ ; $- \bullet> -$ ; $- \bullet> +$

Phép toán ↑ kết hợp phải nên $\uparrow <\bullet \uparrow$

E - E + E ⇒ handle là E - E

E ↑ E ↑ E ⇒ handle là E ↑ E (phần cuối)

**Ví dụ 4.17:** Với chuỗi nhập **id + id * id**

Ta có bảng quan hệ thứ bậc các toán tử như sau :

| | **id** | + | * | **$** |
|---|---|---|---|---|
| **id** | | •> | •> | •> |
| + | <• | •> | <• | •> |
| * | <• | •> | •> | •> |
| **$** | <• | <• | <• | |

Sử dụng $ để đánh dấu cuối chuỗi và $ <• θ, ∀θ.

Ta có chuỗi với các quan hệ thứ bậc được chèn vào là :

$ <• **id** •> + <• **id** •> * <• **id** •> $

Chẳng hạn, <• được chèn vào giữa $ bên trái và id bởi vì <• là mục ở hàng $ và cột id. Handle có thể tìm thấy thông qua quá trình sau :

1. Duyệt chuỗi từ trái sang phải cho đến khi gặp •> đầu tiên (trong ví dụ của chúng ta là •> sau id đầu tiên).

2. Sau đó, duyệt ngược lại (hướng sang trái), vượt qua các $\stackrel{\bullet}{=}$ (nếu có) cho đến khi gặp a <• (trong ví dụ, chúng ta quét ngược về đến $).

3. Handle là chuỗi chứa mọi ký hiệu ở bên trái •> đầu tiên và bên phải <• được gặp trong bước (2), chứa luôn cả các ký hiệu chưa kết thúc xen giữa hoặc bao quanh (trong ví dụ, handle là id đầu tiên).

Với ví dụ trên, sau khi thu gọn id thành E ta được $ E + E * E $.

- Bỏ các ký hiệu chưa kết thúc E ta được $ + * $.
- Thêm các quan hệ ưu tiên ta được $ <• + <• * •> $ . Điều này chứng tỏ handle là E * E được thu gọn thành E.

Vì các ký hiệu chưa kết thúc không ảnh hưởng gì đến việc phân tích cú pháp, nên chúng ta không cần phải phân biệt chúng.

- **Giải thuật 4.5: Phân tích cú pháp thứ bậc toán tử**

**Input:** Chuỗi nhập w và bảng các quan hệ thứ bậc.

**Output:** Nếu w là chuỗi chuẩn dạng đúng thì cho ra một cây phân tích cú pháp. Ngược lại, thông báo lỗi.

**Phương pháp:**

Khởi đầu, Stack chứa ký hiệu $ và bộ đệm chứa câu nhập dạng w$.

Đặt con trỏ ip trỏ tới ký hiệu đầu tiên của w$ ;

**Repeat forever**

**If** $ nằm ở đỉnh Stack và ip chỉ đến $ **then**

**return**

```
Else  begin
    If  a <• b hoặc  a = b then  begin
        Đẩy b vào Stack;
        Dịch ip chỉ đến ký hiệu tiếp theo trong bộ đệm;
    end
    Else if  a •> b  then          /* thu gọn */
        Repeat
            Lấy ký hiệu trên đỉnh Stack ra;
        Until  Ký hiệu kết thúc trên đỉnh Stack có quan hệ  <• với
                ký hiệu kết thúc vừa lấy ra;
        else error ( )
End
```

## VI. BỘ PHÂN TÍCH CÚ PHÁP LR

Phần này giới thiệu một kỹ thuật phân tích cú pháp từ dưới lên khá hiệu quả, có thể sử dụng để phân tích một lớp rộng các văn phạm phi ngữ cảnh. Kỹ thuật này được gọi là **phân tích cú pháp LR(k)**.

**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 nhập được xét 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.

**Các ưu điểm của bộ phân tích cú pháp LR**

- Bộ phân tích cú pháp LR có thể được xây dựng để nhận biết hầu như tất cả các ngôn ngữ lập trình được tạo ra bởi văn phạm phi ngữ cảnh.

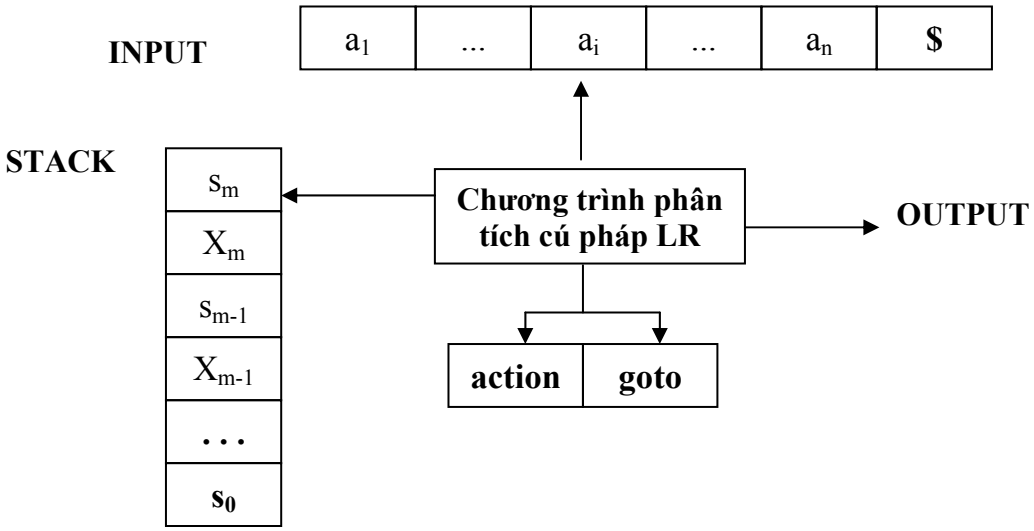
- Phương pháp phân tích cú pháp LR là phương pháp tổng quát của phương pháp chuyên thu gọn không quay lui. Nó có thể được cài đặt có hiệu quả như các phương pháp chuyên thu gọn khác.

- 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.

- Bộ phân tích cú pháp LR cũng có thể xác định lỗi cú pháp nhanh ngay trong khi duyệt dòng nhập từ trái sang phải.

Nhược điểm chủ yếu của phương pháp này là cần phải thực hiện quá nhiều công việc để xây dựng được bộ phân tích cú pháp LR theo kiểu thủ công cho một văn phạm ngôn ngữ lập trình điển hình.

## 1. Thuật toán phân tích cú pháp LR




**Hình 4.12 -** *Mô hình bộ phân tích cú pháp LR*

- **Stack** lưu một chuỗi $s_0X_1s_1X_2s_2 \dots X_ms_m$ trong đó sm nằm trên đỉnh Stack. Xi là một ký hiệu văn phạm, si là một trạng thái tóm tắt thông tin chứa trong Stack bên dưới nó.
- **Bảng phân tích** bao gồm 2 phần: hàm action và hàm goto.
  - ✓ **action[$s_m$, $a_i$]** có thể có một trong 4 giá trị :
    1. **shift s**: đẩy s, trong đó s là một trạng thái.
    2. **reduce A→ β**: thu gọn bằng luật sinh A→ β.
    3. **accept**: Chấp nhận
    4. **error**: Báo lỗi
  - ✓ **Goto** lấy 2 tham số là một trạng thái và một ký hiệu văn phạm, nó sinh ra một trạng thái.

**Cấu hình** (configuration) của một bộ phân tích cú pháp LR là một cặp thành phần, trong đó, thành phần đầu là nội dung của Stack, phần sau là chuỗi nhập chưa phân tích: $(s_0X_1s_1X_2s_2 \dots X_ms_m, a_i\ a_{i+1} \dots a_n\ \$)$

Với sm là ký hiệu trên đỉnh Stack, ai là ký hiệu nhập hiện tại, cấu hình có được sau mỗi dạng bước đẩy sẽ như sau :

1. Nếu **action[$s_m$, $a_i$] = Shift s** : Thực hiện phép đẩy để được cấu hình mới:

$$(s_0X_1s_1X_2s_2 \dots X_ms_m\ a_is,\ a_{i+1} \dots a_n\ \$)$$

Phép đẩy làm cho s nằm trên đỉnh Stack, ai+1 trở thành ký hiệu hiện hành.

2. Nếu **action [$s_m$, $a_i$] = Reduce(A → β)** thì thực hiện phép thu gọn để được cấu hình: $(s_0X_1s_1X_2s_2 \dots X_{m-i}s_{m-i}\ As,\ a_i\ a_{i+1} \dots a_n\ \$)$

Trong đó, s = goto[$s_{m-i}$, A] và r là chiều dài số lượng các ký hiệu của β. Ở đây, trước hết 2r phần tử của Stack sẽ bị lấy ra, sau đó đẩy vào A và s.

3. Nếu **action[$s_m$, $a_i$] = accept**: quá trình phân tích kết thúc.

4. Nếu **action[$s_m$, $a_i$] = error**: gọi thủ tục phục hồi lỗi.

- **Giải thuật 4.6 : Phân tích cú pháp LR**

**Input:** Một chuỗi nhập w, một bảng phân tích LR với hàm action và goto cho văn phạm G.

**Output:** Nếu w ∈ L(G), đưa ra một sự phân tích dưới lên cho w . Ngược lại, thông báo lỗi.

**Phương pháp:**

Khởi tạo s0 là trạng thái khởi tạo nằm trong Stack và w$ nằm trong bộ đệm nhập.

```
Đặt ip vào ký hiệu đầu tiên của w$;
Repeat forever begin
        Gọi s là trạng thái trên đỉnh Stack và a là ký hiệu được trỏ bởi ip;
        If action[s, a] = Shift s' then begin
                Đẩy a và sau đó là s' vào Stack;
                Chuyển ip tới ký hiệu kế tiếp;
        end
        else if action[s, a] = Reduce (A → β) then begin
                Lấy 2 * | β| ký hiệu ra khỏi Stack;
                Gọi s' là trạng thái trên đỉnh Stack;
                Đẩy A, sau đó đẩy goto[s', A] vào Stack;
                Xuất ra luật sinh A → β;
    end
        else if action[s, a] = accept then
                return
        else error ( )
end
```



**Ví dụ 4.18:** Hình sau trình bày các hàm action và goto của bảng phân tích cú pháp LR cho văn phạm của các biểu thức số học dưới đây với các toán tử 2 ngôi + và *

(1) E→ E + T

(2) E→ T

(3) T → T * F

(4) T→ F

| Trạng thái | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | $s_5$ | | | $s_4$ | | | 1 | 2 | 3 |
| 1 | | $s_6$ | | | | acc | | | |

(5) F → (E)

(6) F → **id**

**Ý nghĩa:**

**si : chuyển** trạng thái i

**ri : thu gọn** bởi luật sinh i

**acc**: accept (chấp nhận)

**error :** khoảng trống

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **2** | | $r_2$ | $s_7$ | | $r_2$ | $r_2$ | | | |
| **3** | | $r_4$ | $r_4$ | | $r_4$ | $r_4$ | | | |
| **4** | $s_5$ | | | $s_4$ | | | 8 | 2 | 3 |
| **5** | | $r_6$ | $r_6$ | | $r_6$ | $r_6$ | | | |
| **6** | $s_5$ | | | $s_4$ | | | | 9 | 3 |
| **7** | $s_5$ | | | $s_4$ | | | | | 10 |
| **8** | | $s_6$ | | | $s_{11}$ | | | | |
| **9** | | $r_1$ | $s_7$ | | $r_1$ | $r_1$ | | | |
| **10** | | $r_3$ | $r_3$ | | $r_3$ | $r_3$ | | | |
| **11** | | $r_5$ | $r_5$ | | $r_5$ | $r_5$ | | | |

**Hình 4.13 -** *Bảng phân tích cú pháp SLR cho văn phạm ví dụ*

Với chuỗi nhập **id * id + id**, các bước chuyển trạng thái trên Stack và nội dung bộ đệm nhập được trình bày như sau :

| STACK | INPUT | ACTION |
|---|---|---|
| (1) 0 | id * id + id $ | Shift |
| (2) 0 id 5 | * id + id $ | Reduce by F→ id |
| (3) 0 F 3 | * id + id $ | Reduce by T → F |
| (4) 0 T 2 | * id + id $ | Shift |
| (5) 0 T 2 * 7 | id + id $ | Shift |
| (6) 0 T 2 * 7 id 5 | + id $ | Reduce by F → id |
| (7) 0 T 2 * 7 F 10 | + id $ | Reduce by T → T * F |
| (8) 0 T 2 | + id $ | Reduce by E→ T |
| (9) 0 E 1 | + id $ | Shift |
| (10) 0 E 1 + 6 | id $ | Shift |
| (11) 0 E 1 + 6 id 5 | $ | Reduce by F → id |
| (12) 0 E 1 + 6 F 3 | $ | Reduce by T → F |
| (13) 0 E 1 + 6 T 9 | $ | Reduce by E→ E + T |
| (14) 0 E 1 | $ | **Thành công** |

## 2. Văn phạm LR

Làm thế nào để xây dựng được một bảng phân tích cú pháp LR cho một văn phạm đã cho? Một văn phạm có thể xây dựng được một bảng phân tích cú pháp cho nó được gọi là văn phạm LR. Có những văn phạm phi ngữ cảnh không thuộc loại LR, nhưng

nói chung là ta có thể tránh được những văn phạm này trong hầu hết các kết cấu ngôn ngữ lập trình điển hình.

Có một sự khác biệt rất lớn giữa các văn phạm LL và LR. Để cho một văn phạm là LR(k), chúng ta phải có khả năng nhận diện được sự xuất hiện của vế phải của một luật sinh ứng với k ký hiệu đọc trước. Đòi hỏi này ít khắt khe hơn so với các văn phạm LL(k). Vì vậy, các văn phạm LR có thể mô tả được nhiều ngôn ngữ hơn so với các văn phạm LL.

### 3. Xây dựng bảng phân tích cú pháp SLR

Phần này trình bày cách xây dựng một bảng phân tích cú pháp LR từ văn phạm. Chúng ta sẽ đưa ra 3 phương pháp khác nhau về tính hiệu quả cũng như tính dễ cài đặt. Phương pháp thứ nhất được gọi là "LR đơn giản" (Simple LR - SLR), là phương pháp "yếu" nhất nếu tính theo số lượng văn phạm có thể xây dựng thành công bằng phương pháp này, nhưng đây lại là phương pháp dễ cài đặt nhất. Ta gọi bảng phân tích cú pháp tạo ra bởi phương pháp này là bảng SLR và bộ phân tích cú pháp tương ứng là bộ phân tích cú pháp SLR, với văn phạm tương đương là văn phạm SLR.

**a. Mục** (Item): Cho một văn phạm G, mục LR(0) văn phạm là một luật sinh của G với một dấu chấm mục tại vị trí nào đó trong vế phải.

**Ví dụ 4.19:** Luật sinh $A \rightarrow XYZ$ có 4 mục như sau :

$A \rightarrow \bullet XYZ$

$A \rightarrow X\bullet YZ$

$A \rightarrow XY\bullet Z$

$A \rightarrow XYZ\bullet$

Luật sinh $A \rightarrow \varepsilon$ chỉ tạo ra một mục $A \rightarrow \bullet$

**b. Văn phạm tăng cường** (Augmented Grammar)

Giả sử G là một văn phạm với ký hiệu bắt đầu S, ta thêm một ký hiệu bắt đầu mới S' và luật sinh $S' \rightarrow S$ để được văn phạm mới G' gọi là văn phạm tăng cường.

**c. Phép toán bao đóng** (Closure)

Giả sử I là một tập các mục của văn phạm G thì bao đóng closure(I) là tập các mục được xây dựng từ I theo qui tắc sau:

1. Tất cả các mục của I được thêm cho closure(I).
2. Nếu $A \rightarrow \alpha \bullet B\beta \in$ closure(I) và $B \rightarrow \gamma$ là một luật sinh thì thêm $B \rightarrow \bullet \gamma$ vào closure(I) nếu nó chưa có trong đó. Lặp lại bước này cho đến khi không thể thêm vào closure(I) được nữa.

**Ví dụ 4.20:** Xét văn phạm tăng cường của biểu thức:

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

F → (E) | **id**

Nếu I là tập hợp chỉ gồm văn phạm { E'→ • E } thì closure(I) bao gồm:

E' → • E (Luật 1)

E → • E + T (Luật 2)

E → • T (Luật 2)

T → • T * F (Luật 2)

T → • F (Luật 2)

F → • (E) (Luật 2)

F → • id (Luật 2)

**Chú ý rằng:** Nếu một B - luật sinh được đưa vào closure(I) với dấu chấm mục nằm ở đầu vế phải thì tất cả các B - luật sinh đều được đưa vào.

**d. Phép toán Goto**

Goto(I, X), trong đó I là một tập các mục và X là một ký hiệu văn phạm là bao đóng của tập hợp các mục A → αX•β sao cho A → α•Xβ ∈ I.

Cách tính goto(I, X):

1. Tạo một tập I' = ∅.

2. Nếu A → α•Xβ ∈ I thì đưa A→ αX•β vào I', tiếp tục quá trình này cho đến khi xét hết tập I.

3. Goto(I, X) = closure(I')

**Ví dụ 4.21:** Giả sử I = { E' → E•, E → E • + T }.

Tính goto (I, +) ?

Ta có I' = { E→ E + • T }

( goto (I, +) = closure(I') bao gồm các mục :

E → E + • T (Luật 1)

T → • T * F (Luật 2)

T → • F (Luật 2)

F → • (E) (Luật 2)

F → • id (Luật 2)

**e. Giải thuật xây dựng họ tập hợp các mục LR(0) của văn phạm G'**

Gọi C là họ tập hợp các mục LR(0) của văn phạm tăng cường G'. Ta có thủ tục xây dựng C như sau :

**Procedure** Item (G')

**begin**

C := closure({ S' → •S});

***Repeat***

**For** Với mỗi tập các mục I trong C và mỗi ký hiệu văn phạm X sao cho goto (I, X) $\neq \varnothing$ và goto(I, X) $\notin$ C do

Thêm goto(I, X) vào C;

***Until*** Không còn tập hợp mục nào có thể thêm vào C;

**end;**

**Ví dụ 4.22:** Với văn phạm tăng cường G' của biểu thức trong ví dụ 4.20 :

Ta xây dựng họ tập hợp mục C của văn phạm như sau:

Closure({E'→ E}):

**$I_0$** : E'→ • E
E → • E + T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id

Goto ($I_0$, E) **$I_1$**: E'→ E •
E → E • + T

Goto ($I_0$, T) **$I_2$**: E → T •
T → T • * F

Goto ($I_0$, F) **$I_3$**: T → F •

Goto ($I_0$, **(** ) **$I_4$**: F → (• E)
E → • E + T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id

Goto ($I_1$, +) **$I_6$**: E → E + • T
T → • T * F
T → • F
F → • (E)
F → • id

Goto ($I_2$, *) **$I_7$**: T → T* • F
F → • (E)
F → • id

Goto ($I_4$, E) **$I_8$**: T → (E •)
E → E • + T

Goto ($I_6$,T) **$I_9$**: E → E + T •
T → T • * F

Goto ($I_7$,F) **$I_{10}$**: T → T * F •

Goto ($I_8$,**)** ) **$I_{11}$**: F → (E) •



**f. Thuật toán xây dựng bảng phân tích SLR**

- **Giải thuật 4.7: Xây dựng bảng phân tích SLR**

**Input:** Một văn phạm tăng cường G'

**Output:** Bảng phân tích SLR với hàm action và goto

**Phương pháp:**

1. Xây dựng C = { $I_0, I_1, ..., I_n$ }, họ tập hợp các mục LR(0) của G'.

2. Trạng thái i được xây dựng từ Ii .Các action tương ứng trạng thái i được xác định như sau:

2.1. Nếu $A \to \alpha \bullet a\beta \in I_i$ và goto ($I_i$, a) = Ij thì action[i, a] = "**shift j**". Ở đây a là ký hiệu kết thúc.

2.2. Nếu $A \to \alpha\bullet \in I_i$ thì action[i, a] = "**reduce (A → α)**", ∀a ∈ FOLLOW(A). Ở đây A không phải là S'

2.3. Nếu $S' \to S\bullet \in I_i$ thì action[i, $] = "**accept**".

Nếu một action đụng độ được sinh ra bởi các luật trên, ta nói văn phạm không phải là SLR(1). Giải thuật sinh ra bộ phân tích cú pháp sẽ thất bại trong trường hợp này.

3. Với mọi ký hiệu chưa kết thúc A, nếu goto ($I_i$,A) = $I_j$ thì goto [i, A] = j

4. Tất cả các ô không xác định được bởi 2 và 3 đều là "**error**"

5. Trạng thái khởi đầu của bộ phân tích cú pháp được xây dựng từ tập các mục chứa S'→ • S

**Ví dụ 4.23:** Ta xây dựng bảng phân tích cú pháp SLR cho văn phạm tăng cường G' trong ví dụ 4.20 trên.

| | | |
|---|---|---|
| E' → E | (0) E'→ E | (4) T → F |
| E → E + T \| T | (1) E → E + T | (5) F → **(**E**)** |
| T → T * F \| F | (2) E → T | (6) F → **id** |
| F → **(**E**)** \| **id** | (3) T → T * F | |

1. C = { $I_0, I_1, ... I_{11}$ }

2. FOLLOW(E) = {+, ), $}

FOLLOW(T) = {*, +, ), $}

FOLLOW(F) = {*, +, ), $}

Dựa vào họ tập hợp mục C đã được xây dựng trong ví dụ 4.22, ta thấy:

Trước tiên xét tập mục $I_0$ : Mục F → • (E) cho ra action[0, **(**] = "shift 4", và mục F → • **id** cho action[0, id] = "shift 5". Các mục khác trong $\mathbf{I_0}$ không sinh được hành động nào.

Bây giờ xét $\mathbf{I_1}$ : Mục E'→ E • cho action[1, $] = "accept", mục E → E • + T cho action[1, +] = "shift 6".

Kế đến xét $\mathbf{I_2}$ : E → T •

$T \rightarrow T \bullet * F$

Vì FOLLOW(E) = {+, ), $}, mục đầu tiên làm cho action[2, $] = action[2,+] = "reduce (E ( T)". Mục thứ hai làm cho action[2,*] = "shift 7".

Tiếp tục theo cách này, ta thu được bảng phân tích cú pháp SLR đã trình bày trong hình 4.13.

Bảng phân tích xác định bởi giải thuật 4.7 gọi là bảng SLR(1) của văn phạm G, bộ phân tích LR sử dụng bảng SLR(1) gọi là bộ phân tích SLR(1) và văn phạm có một bảng SLR(1) gọi là văn phạm SLR(1).

Mọi văn phạm SLR(1) đều không mơ hồ. Tuy nhiên có những văn phạm không mơ hồ nhưng không phải là SLR(1).

**Ví dụ 4.24:** Xét văn phạm G với tập luật sinh như sau:

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow * R$

$L \rightarrow$ **id**

$R \rightarrow L$

Đây là một văn phạm không mơ hồ nhưng không phải là văn phạm SLR(1).

Họ tập hợp các mục C bao gồm:

$\mathbf{I_0}$ : $S' \rightarrow \bullet S$

$S \rightarrow \bullet L = R$

$S \rightarrow \bullet R$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet$ id

$R \rightarrow \bullet L$

$\mathbf{I_1}$ : $S' \rightarrow S \bullet$

$\mathbf{I_2}$ : $S \rightarrow L \bullet = R$

$R \rightarrow L \bullet$

$\mathbf{I_3}$ : $S \rightarrow R \bullet$

$\mathbf{I_4}$ : $L \rightarrow * \bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet$ id

$\mathbf{I_5}$ : $L \rightarrow$ id $\bullet$

$\mathbf{I_6}$ : $S \rightarrow L = \bullet R$

$R \rightarrow \bullet L$

$L \rightarrow \bullet * R$

$L \rightarrow \bullet$ id

$\mathbf{I_7}$ : $L \rightarrow * R \bullet$

$\mathbf{I_8}$ : $R \rightarrow L \bullet$

$\mathbf{I_9}$ : $S \rightarrow L = R \bullet$

Khi xây dựng bảng phân tích SLR cho văn phạm, khi xét tập mục $I_2$ ta thấy mục đầu tiên trong tập này làm cho action[2, =] = "shift 6". Bởi vì = ∈ FOLLOW(R), nên mục thứ hai sẽ đặt action[2, =] = "reduce (R → L)" ⇒ Có sự đụng độ tại action[2, =]. Vậy văn phạm trên không là văn phạm SLR(1).

### 4. Xây dựng bảng phân tích cú pháp LR chính tắc (Canonical LR Parsing Table)

**a. Mục LR(1)** của văn phạm G là một cặp dạng [A → α•β, a], trong đó A → αβ là luật sinh, a là một ký hiệu kết thúc hoặc $.

**b. Thuật toán xây dựng họ tập hợp mục LR(1)**

- **Giải thuật 4.8: Xây dựng họ tập hợp các mục LR(1)**

**Input :** Văn phạm tăng cường G'

**Output:** Họ tập hợp các mục LR(1).

**Phương pháp:** Các thủ tục closure, goto và thủ tục chính Items như sau:

```
Function Closure (I);
begin
    Repeat
      For Mỗi mục [A → α•Bβ,a] trong I, mỗi luật sinh B → γ trong G' và mỗi
          ký hiệu kết thúc b ∈ FIRST (βa) sao cho [B → • γ, b] ∉ I do
          Thêm [B → •γ, b] vào I;
      Until Không còn mục nào có thể thêm cho I được nữa;
      return I;
end;

Function goto (I, X);
begin
      Gọi J là tập hợp các mục [A → αX•β, a] sao cho [A → α•Xβ, a]∈ I;
      return Closure(J);
end;

Procedure Items (G');
begin
      C := Closure ({[S' → •S, $]})
      Repeat
      For Mỗi tập các mục I trong C và mỗi ký hiệu văn phạm X
          sao cho goto(I, X) ≠ ∅ và goto(I, X) ∉ C do
```

Thêm goto(I, X) vào C;

***Until*** Không còn tập các mục nào có thể thêm cho C;

**end;**

**Ví dụ 4.25:** Xây dựng bảng LR chính tắc cho văn phạm tăng cường G' có chứa các luật sinh như sau :

S' → S

(1) S → L = R

(2) S → R

(3) L → * R

(4) L → **id**

(5) R → L

Trong đó: tập ký hiệu chưa kết thúc ={S, L, R} và tập ký hiệu kết thúc {=, *, id, $}

**$I_0$** : S' → • S, $

Closure (S' → •S, $) S → • L = R, $

S → • R, $

L → • * R, = | $

L → • id, = | $

R → • L, $

Goto ($I_0$,S) **$I_1$** : S' → S •, $

Goto ($I_0$, L) **$I_2$** : S → L • = R, $

R → L •, $

Goto ($I_0$,R) **$I_3$**: S → R •, $

Goto ($I_0$,*) **$I_4$**: L → * • R, = | $

R → • L, = | $

L → • * R, = | $

R → • id, = | $

Goto ($I_0$,id) **$I_5$** : L → id •, = | $

Goto ($I_4$,R) **$I_7$** : L → * R•, = | $

Goto ($I_4$, L) **$I_8$** : R→ L•, = | $

Goto ($I_6$,R) **$I_9$** : S → L = R•, $

Goto ($I_6$,L) **$I_{10}$** : R → L•, $

Goto ($I_6$,*) **$I_{11}$** : L → * • R, $

R → • L, $

L → • * R, $

R → • id, $

Goto ($I_6$, id) **$I_{12}$** : L → id •, $

Goto ($I_{11}$,R) **$I_{13}$** : R → * R•, $

Goto ($I_{11}$,L) ≡ **$I_{10}$**

Goto ($I_2$,=) $\mathbf{I_6}$ : S → L = • R, $ Goto ($I_{11}$,*) ≡ $\mathbf{I_{11}}$

R → • L, $

L → • * R, $ Goto ($I_{11}$,id) ≡ $I_{12}$

L → • id, $

**c. Thuật toán xây dựng bảng phân tích cú pháp LR chính tắc**

- **Giải thuật 4.9: Xây dựng bảng phân tích LR chính tắc**

**Input:** Văn phạm tăng cường G'

**Output:** Bảng LR với các hàm action và goto

**Phương pháp:**

1. Xây dựng C = { $I_0$, $I_1$, .... $I_n$ } là họ tập hợp mục LR(1)

2. Trạng thái thứ i được xây dựng từ $I_i$. Các action tương ứng trạng thái i được xác định như sau:

2.1. Nếu [A → α • aβ,b] ∈ $I_i$ và goto($I_i$,a) = Ij thì action[i, a]= "**shift j**". Ở đây a phải là ký hiệu kết thúc.

2.2. Nếu [A → α •, a]∈ $I_i$ , A ∈ S' thì action[i, a] = " **reduce (A → α)**"

2.3. Nếu [S' → S•,$] ∈ $I_i$ thì action[i, $] = "**accept**".

Nếu có một sự đụng độ giữa các luật nói trên thì ta nói văn phạm không phải là LR(1) và giải thuật sẽ thất bại.

3. Nếu goto($I_i$, A) = $I_j$ thì goto[i,A] = j

4. Tất cả các ô không xác định được bởi 2 và 3 đều là "**error**"

5. Trạng thái khởi đầu của bộ phân tích cú pháp được xây dựng từ tập các mục chứa [S' → •S,$]

Bảng phân tích xác định bởi giải thuật 4.9 gọi là bảng phân tích LR(1) chính tắc của văn phạm G, bộ phân tích LR sử dụng bảng LR(1) gọi là bộ phân tích LR(1) chính tắc và văn phạm có một bảng LR(1) không có các action đa trị thì được gọi là văn phạm LR(1).

**Ví dụ 4.26:** Xây dựng bảng phân tích LR chính tắc cho văn phạm ở ví dụ trên

| Trạng thái | Action | | | | Goto | | |
|---|---|---|---|---|---|---|---|
| | = | * | id | $ | S | L | R |
| **0** | | $s_4$ | $s_5$ | | 1 | 2 | 3 |
| **1** | | | | acc | | | |
| **2** | $s_6$ | | | $r_5$ | | | |
| **3** | | | | $r_2$ | | | |
| **4** | | $s_4$ | $s_5$ | | | 8 | 7 |
| **5** | $r_4$ | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **6** | | $s_{11}$ | $s_{12}$ | | | 10 | 9 |
| **7** | $r_3$ | | | | | | |
| **8** | $r_5$ | | | | | | |
| **9** | | | | $r_1$ | | | |
| **10** | | | | $r_5$ | | | |
| **11** | | $s_{11}$ | $s_{12}$ | | | 10 | 13 |
| **12** | | | | $r_4$ | | | |
| **13** | | | | $r_3$ | | | |

**Hình 4.14 -** *Bảng phân tích cú pháp LR chính tắc*

Mỗi văn phạm SLR(1) là một văn phạm LR(1), nhưng với một văn phạm SLR(1), bộ phân tích cú pháp LR chính tắc có thể có nhiều trạng thái hơn so với bộ phân tích cú pháp SLR cho văn phạm đó.

## 5. Xây dựng bảng phân tích cú pháp LALR

Phần này giới thiệu phương pháp cuối cùng để xây dựng bộ phân tích cú pháp LR - kỹ thuật LALR (Lookahead-LR), phương pháp này thường được sử dụng trong thực tế bởi vì những bảng LALR thu được nói chung là nhỏ hơn nhiều so với các bảng LR chính tắc và phần lớn các kết cấu cú pháp của ngôn ngữ lập trình đều có thể được diễn tả thuận lợi bằng văn phạm LALR.



**a. Hạt nhân (core)** của một tập hợp mục LR(1)

1. Một tập hợp mục LR(1) có dạng {[ $A \rightarrow \alpha \bullet \beta$, a]}, trong đó $A \rightarrow \alpha\beta$ là một luật sinh và a là ký hiệu kết thúc có hạt nhân (core) là tập hợp $\{A \rightarrow \alpha \bullet \beta\}$.

2. Trong họ tập hợp các mục LR(1) C = { $I_0$, $I_1$, ..., $I_n$ } có thể có các tập hợp các mục có chung một hạt nhân.

**Ví dụ 4.27:** Trong ví dụ 4.25, ta thấy trong họ tập hợp mục có một số các mục có chung hạt nhân là :

$I_4$ và $I_{11}$

$I_5$ và $I_{12}$

$I_7$ và $I_{13}$

$I_8$ và $I_{10}$



**b. Thuật toán xây dựng bảng phân tích cú pháp LALR**

- **Giải thuật 4.10: Xây dựng bảng phân tích LALR**

**Input:** Văn phạm tăng cường G'

**Output:** Bảng phân tích LALR

**Phương pháp:**

1. Xây dựng họ tập hợp các mục LR(1) C = { $I_0$, $I_1$, ..., $I_n$ }

2. Với mỗi hạt nhân tồn tại trong tập các mục LR(1) tìm trên tất cả các tập hợp có cùng hạt nhân này và thay thế các tập hợp này bởi hợp của chúng.

3. Đặt C' = { $I_0, I_1, ..., I_m$ } là kết quả thu được từ C bằng cách hợp các tập hợp có cùng hạt nhân. Action tương ứng với trạng thái i được xây dựng từ Ji theo cách thức như giải thuật 4.9.

Nếu có một sự đụng độ giữa các action thì giải thuật xem như thất bại và ta nói văn phạm không phải là văn phạm LALR(1).

4. Bảng goto được xây dựng như sau

Giả sử $J = I_1 \cup I_2 \cup ... \cup I_k$. Vì $I_1, I_2, ... I_k$ có chung một hạt nhân nên goto $(I_1,X)$, goto $(I_2,X)$, ..., goto $(I_k,X)$ cũng có chung hạt nhân. Đặt K bằng hợp tất cả các tập hợp có chung hạt nhân với goto $(I_1,X) \Rightarrow$ goto(J, X) = K.

**Ví dụ 4.28:** Với ví dụ trên, ta có họ tập hợp mục C' như sau
C' = {$I_0, I_1, I_2, I_3, I_{411}, I_{512}, I_6, I_{713}, I_{810}, I_9$ }

**$I_0$** : S' → • S, $
closure (S' → •S, $) : S → • L = R, $
S → • R, $
L → • * R, =
L → • id, =
R → • L, $

**$I_1$** = Goto ($I_0$,S) : S' → S •, $

**$I_2$** = Goto ($I_0$, L) : S → L • = R, $
R → L •, $

**$I_3$** = Goto ($I_0$,R) : S → R •

**$I_{411}$** = Goto ($I_0$,*), Goto ($I_6$,*) :
L → * • R, = | $
R → • L, = | $
L → • * R, = | $
R → • id, = | $

**$I_{512}$** = Goto ($I_0$,id), Goto ($I_6$,id) :
L → id •, = | $

**$I_6$** = Goto($I_2$,=) :
S → L = • R,$
R → • L, $
L → • * R, $
L → • id, $

**$I_{713}$** = Goto($I_{411}$, R) :
L → * R•, = | $

**$I_{810}$** = Goto($I_{411}$, L), Goto($I_6$, L):
R → L•, = | $

**$I_9$** = Goto($I_6$, R) :
S → L = R•, $

Ta có thể xây dựng bảng phân tích cú pháp LALR cho văn phạm như sau:

| State | Action | | | | Goto | | |
|---|---|---|---|---|---|---|---|
| | = | * | id | $ | S | L | R |
| **0** | | $s_{411}$ | $s_{512}$ | | 1 | 2 | 3 |
| **1** | | | **acc** | | | | |
| **2** | $s_6$ | | | | | | |
| **3** | | | | $r_2$ | | | |
| **411** | | | | | | 810 | 713 |
| **512** | $r_4$ | | | $r_4$ | | | |
| **6** | | $s_{411}$ | $s_{512}$ | | | 810 | 9 |
| **713** | $r_3$ | | | $r_3$ | | | |
| **810** | $r_5$ | | | $r_5$ | | | |
| **9** | | | | $r_1$ | | | |

**Hình 4.15 -** *Bảng phân tích cú pháp LALR*

Bảng phân tích được tạo ra bởi giải thuật 4.10 gọi là bảng phân tích LALR cho văn phạm G. Nếu trong bảng không có các action đụng độ thì văn phạm đã cho gọi là văn phạm LALR(1). Họ tập hợp mục C' được gọi là họ tập hợp mục LALR(1).

## VII. SỬ DỤNG CÁC VĂN PHẠM MƠ HỒ

Như chúng ta đã nói trước đây rằng mọi văn phạm mơ hồ đều không phải là LR. Tuy nhiên có một số văn phạm mơ hồ lại rất có ích cho việc đặc tả và cài đặt ngôn ngữ. Chẳng hạn văn phạm mơ hồ cho kết cấu biểu thức đặc tả được một cách ngắn gọn và tự nhiên hơn bất kỳ một văn phạm không mơ hồ nào khác. Văn phạm mơ hồ còn được dùng trong việc tách biệt các kết cấu cú pháp thường gặp cho quá trình tối ưu hóa. Với một văn phạm mơ hồ, chúng ta có thể đưa thêm các luật sinh mới vào văn phạm.

Mặc dù các văn phạm là đa nghĩa, trong mọi trường hợp, chúng ta sẽ đưa thêm các quy tắc khử mơ hồ (disambiguating rule), để chỉ cho phép chọn một cây phân tích cú pháp cho mỗi một câu nhập. Theo cách này, đặc tả ngôn ngữ về tổng thể vẫn là đơn nghĩa.

### 1. Sử dụng độ ưu tiên và tính kết hợp của các toán tử để giải quyết đụng độ.

Xét văn phạm của biểu thức số học với hai toán tử + và * :

E → E + E | E * E | (E) |**id** **(1)**

Đây là một văn phạm mơ hồ vì nó không xác định độ ưu tiên và tính kết hợp của các tóan tử + và *. Trong khi đó ta có văn phạm tương đương không mơ hồ cho biểu thức có dạng như sau:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$ **(2)**

$F \rightarrow (E) \mid \mathbf{id}$

Văn phạm này xác định rằng + có độ ưu tiên thấp hơn * và cả hai toán tử đều kết hợp trái. Tuy nhiên có 2 lý do để chúng ta sử dụng văn phạm (1) chứ không phải là (2):

1. Dễ dàng thay đổi tính kết hợp và độ ưu tiên của + và * mà không phá hủy các luật sinh và số các trạng thái của bộ phân tích (như ta sẽ thấy sau này).

2. Bộ phân tích cho văn phạm (2) sẽ mất thời gian thu gọn bởi các luật sinh $E \rightarrow T$ và $T \rightarrow F$. Hai luật sinh này không nói lên được tính kết hợp và độ ưu tiên.

Nhưng với văn phạm (1) thì làm thế nào để tránh sự đụng độ? Trước hết chúng ta hãy thành lập bộ sưu tập C các tập mục LR(0) của văn phạm tăng cường của nó.

$\mathbf{I_0}$: $E' \rightarrow \bullet E$
$E \rightarrow \bullet E + E$
$E \rightarrow \bullet E * E$
$E \rightarrow \bullet (E)$
$E \rightarrow \bullet id$

$Goto(I_0,E)$ $\mathbf{I_1}$: $E' \rightarrow E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

$Goto(I_0,()$ $\mathbf{I_2}$: $E \rightarrow (\bullet E)$
$E \rightarrow \bullet E + E$
$E \rightarrow \bullet E * E$
$E \rightarrow \bullet (E)$
$E \rightarrow \bullet id$

$Goto(I_0,id)$ $\mathbf{I_3}$: $E \rightarrow id \bullet$

$Goto(I_1,+)$ $\mathbf{I_4}$: $E \rightarrow E + \bullet E$
$E \rightarrow \bullet E + E$
$E \rightarrow \bullet E * E$
$E \rightarrow \bullet (E)$

$Goto(I_2,E)$ $\mathbf{I_6}$: $E' \rightarrow (E \bullet)$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

$Goto(I_2,() \equiv I_2$

$Goto(I_2,id) \equiv I_3$

$Goto(I_4,E)$ $\mathbf{I_7}$: $E \rightarrow E + E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

$Goto(I_4,( ) \equiv I_2$

$Goto(I_4,id) \equiv I_3$

$Goto(I_5,E)$ $\mathbf{I_8}$: $E \rightarrow E * E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

$Goto(I_5,() \equiv I_2$

$Goto(I_5,id) \equiv I_3$

$Goto(I_6,))$ $\mathbf{I_9}$: $E \rightarrow (E) \bullet$

$Goto(I_6,+) \equiv I_4$

$Goto(I_6,*) \equiv I_5$

| | | | |
|---|---|---|---|
| | | $E \rightarrow \bullet$ id | $Goto(I_7,+) \equiv I_4$ |
| | | | $Goto(I_7,*) \equiv I_5$ |
| $Goto(I_1,*)$ | $\mathbf{I_5}$: | $E \rightarrow E * \bullet E$ | $Goto(I_8,+) \equiv I_4$ |
| | | $E \rightarrow \bullet E + E$ | $Goto(I_8,*) \equiv I_5$ |
| | | $E \rightarrow \bullet E * E$ | |
| | | $E \rightarrow \bullet ( E)$ | |
| | | $E \rightarrow \bullet$ id | |

Bảng phân tích SLR đụng độ được xây dựng như sau :

| **Trạng thái** | **Action** | | | | | | **Goto** |
|---|---|---|---|---|---|---|---|
| | **id** | **+** | ***** | **(** | **)** | **$** | **E** |
| **0** | $s_3$ | | | $s_2$ | | | 1 |
| **1** | | $s_4$ | $s_5$ | | | **acc** | |
| **2** | $s_3$ | | | | | | 6 |
| **3** | | $r_4$ | $r_4$ | | $r_4$ | $r_4$ | |
| **4** | $s_3$ | | | $s_2$ | | | 7 |
| **5** | $s_3$ | | | $s_2$ | | | 8 |
| **6** | | $s_4$ | $s_5$ | | $s_9$ | | |
| **7** | | $s_4$ / $r_1$ | $s_5$ / $r_1$ | | $r_1$ | $r_1$ | |
| **8** | | $s_4$ / $r_2$ | $s_5$ / $r_2$ | | $r_2$ | $r_2$ | |
| **9** | | $r_3$ | $r_3$ | | $r_3$ | $r_3$ | |

**Hình 4.16 -** *Bảng phân tích cú pháp SLR đụng độ*

Nhìn vào bảng SLR trong hình trên, ta thấy có sụ đụng độ tại action [7, +] và action [7,*]; action [8, +] và action [8,*].

Chúng ta sẽ giải quyết sự đụng độ này bằng quy tắc kết hợp và độ ưu tiên của các toán tử. Xét chuỗi nhập **id + id * id**

| **Stack** | **Input** | **Output** |
|---|---|---|
| 0 | id + id * **id** $ | |
| 0 id 3 | + id * id $ | Shift $s_3$ |
| 0 E 1 | + id * id $ | Reduce by E → id |
| 0 E 1 + 4 | id * id $ | Shift $s_4$ |
| 0 E 1 + 4 id 3 | * id $ | Shift $s_3$ |
| 0 E 1 + 4 E 7 | * id $ | Reduce by E → id |

| | | |
|---|---|---|
| | | |

Bây giờ đến ô đụng độ action[7, *] nên lấy r1 hay s5? Lúc này chúng ta đã phân tích qua phần chuỗi id * id. Nếu ta chọn r1 tức là thu gọn bởi luật sinh E → E + E, có nghĩa là chúng ta đã thực hiện phép cộng trước. Do vậy nếu ta muốn tóan tử * có độ ưu tiên cao hơn + thì phải chọn $s_5$.

Nếu chuỗi nhập là id + id + id thì quá trình phân tích văn phạm dẫn đến hình trạng hiện tại là :

| **Stack** | **Output** |
|---|---|
| 0 E 1 + 4 E 7 | + id $ |

Sẽ phải xét action [7, +] nên chọn r1 hay s4? Nếu ta chọn r1 tức là thu gọn bởi luật sinh E → E + E tức là + thực hiện trước hay toán tử + có kết hợp trái => action [7, +] = r1

Một cách tương tự nếu ta quy định phép * có độ ưu tiên cao hơn + và phép * **kết hợp trái** thì action [8, *] = r2 vì * kết hợp trái (xét chuỗi id * id * id). Action [8,+] = r2 vì toán tử * có độ ưu tiên cao hơn + (trường hợp xét chuỗi id * id + id)

Sau khi đã giải quyết được sự đụng độ đó ta có được bảng phân tích SLR đơn giản hơn bảng phân tích của văn phạm tương đương (2) (chỉ sử dụng 10 trạng thái thay vì 12 trạng thái). Tương tự, ta có thể xây dựng bảng phân tích LR chính tắc và LALR cho văn phạm (1).

## 2. Giải quyết trường hợp văn phạm mơ hồ IF THEN ELSE

Xét văn phạm cho lệnh điều kiện:

Stmt → **if** expr **then** stmt **else** stmt
| **if** expr **then** stmt
| **other**

Để đơn giản ta viết **i** thay cho **if expr then**, **S** thay cho **stmt**, **e** thay cho **else** và a thay cho other, ta có văn phạm viết lại như sau :

S' → S
S → iS eS (1)
S → iS (2)
S → a (3)

Họ tập hợp mục C các tập mục LR(0) là:

$\mathbf{I_0}$ : S' → • S,
S → • iSeS
S → • iS
S → • a

Goto ($I_0$,S) $\mathbf{I_1}$ : S' → S •

Goto ($I_0$,i) $\mathbf{I_2}$ : S → i • SeS
S → i • S
S → • iSeS
S → • iS
S → • a

Goto ($I_0$,a) $\mathbf{I_3}$: S → a •

Goto ($I_2$, S) $\mathbf{I_4}$: S → iS• eS
S → iS•

Goto ($I_4$,e) $\mathbf{I_5}$ : S → iSe• S
S → • iSeS
S → • iS
S → • a

Goto ($I_5$,S) $\mathbf{I_6}$ : S → iSeS•
Goto($I_2$,i) ≡ $I_2$
Goto($I_2$,a) ≡ $I_3$
Goto($I_5$,i) ≡ $I_2$
Goto($I_5$,a) ≡ $I_3$

Ta xây dựng bảng phân tích SLR đụng độ như sau:

| **Trạng thái** | **Action** | | | | **Goto** |
|---|---|---|---|---|---|
| | **i** | **e** | **a** | **$** | **S** |
| **0** | $s_2$ | | $s_3$ | | 1 |
| **1** | | | | **acc** | |
| **2** | $s_2$ | | $s_3$ | | 4 |
| **3** | | $r_3$ | | $r_3$ | |
| **4** | | $s_5$\| $r_2$ | | $r_2$ | |
| **5** | $s_2$ | | $s_3$ | | 6 |
| **6** | | $r_1$ | | | |

**Hình 4.17** - *Bảng phân tích cú pháp LR cho văn phạm if - else*

Để giải quyết đụng độ tại action[4, e]. Trường hợp này xảy ra trong tình trạng chuỗi ký hiệu if expr then stmt nằm trong Stack và else là ký hiệu nhập hiện hành. Sử dụng nguyên tắc kết hợp mỗi else với một then chưa kết hợp gần nhất trước đó nên ta phải Shift else vào Stack để kết hợp với **then nên** action [4, e] = $s_5$.

**Ví dụ 4.29:** Với chuỗi nhập **i i a e a**

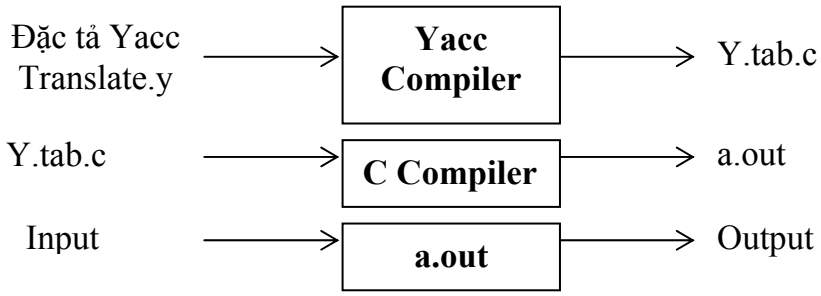
(**if** expr1 **then** **if** expr2 **then** $a_1$ **else** $a_2$)

| Stack | Input | Output |
|---|---|---|
| 0 | i i a e a $ | |
| 0 i 2 | i a e a $ | Shift $s_2$ |
| 0 i 2 i 2 | a e a $ | Shift $s_2$ |
| 0 i 2 i 2 a 3 | e a $ | Shift $s_3$ |
| 0 i 2 i 2 S 4 | a $ | Reduce by S → a |
| 0 i 2 i 2 S 4 e 5 | $ | Shift $s_5$ |
| 0 i 2 i 2 S 4 e 5 a 3 | $ | Shift $s_3$ |
| 0 i 2 i 2 S 4 e 5 S 6 | $ | Reduce by S → a |
| 0 i 2 S 4 | $ | Reduce by S → iS eS |
| 0 s 1 | $ | Reduce by S → iS |

## VIII. BỘ SINH BỘ PHÂN TÍCH CÚ PHÁP

Phần này trình bày cách dùng một bộ sinh bộ phân tích cú pháp (parser generator) hỗ trợ cho việc xây dựng kỳ đầu của một trình biện dịch. Một trong những bộ sinh bộ phân tích cú pháp là YACC (Yet Another Compiler - Compiler). Phiên bản đầu tiên của Yacc được S.C.Johnson tạo ra và hiện Yacc được cài đặt như một lệnh của hệ UNIX và đã được dùng để cài đặt cho hàng trăm trình biên dịch.

### 1. Bộ sinh thể phân tích cú pháp Yacc




**Hình 4.18 -** *Tạo một chương trình dịch input / output với Yacc*

Một chương trình dịch có thể được xây dựng nhờ Yacc bằng phương thức được minh họa trong hình 4.18 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** của hệ UNIX sẽ biến đổi tập tin translate.y thành một chương trình C có tên là **y.tab.C** bằng phương pháp LALR đã trình bày ở trên. Chương trình **y.tab.C** là một biểu diễn của bộ phân tích cú pháp LALR được viết bằng ngôn ngữ C cùng với các thủ tục C khác có thể do người sử dụng chuẩn bị. Bằng cách dịch **y.tab.C** cùng với thư viện **ly** chứa chương trình phân tích cú pháp LR 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.

## 2. Đặc tả YACC

Một chương trình nguồn Yacc bao gồm 3 phần:

**Phần khai báo**

**% %**

**Các luật dịch**

**%%**

**Các thủ tục**

**Ví dụ 4.30:** Để minh họa việc chuẩn bị một chương trình nguồn Yacc, chúng ta hãy xây dựng một chương trình máy tính bỏ túi đơn giản, đọc một biểu thức số học, ước lượng rồi in ra giá trị số của nó. Chúng ta xây dựng bắt đầu từ văn phạm sau :

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \textbf{digit}$$

Token **digit** là một ký hiệu số từ 0 đến 9. Một chương trình Yacc dành cho văn phạm này như sau :



```
%{
# include < ctype.h>
%}
% token DIGIT

%%

line : expr       '\n'              { print ("%d\n", $1); }
;
expr : expr       '+'    term       { $$ = $1 + $3; }
|      term
;
term : term       '* '   factor     { $$ = $1 *  $3; }
|      factor
;
factor:      '('  expr  ')'         { $$ = $2 ; }
|      DIGIT
;
```

```
%%


yylex ( )
{       int c
        c = getchar ( );
        if  (isdigit(c))
        {       yyval = c -'0';
                return DIGIT;
        }
        return c;
 }
```

**Phần khai báo** có thể bao gồm 2 phần nhỏ:

- **Khai báo C** đặt nằm trong cặp dấu %{ và }%. Những khai báo này sẽ được sử dụng trong phần 2 và phần 3.
- **Khai báo các token** (DIGIT là một token). Các token khai báo ở đây sẽ được dùng trong phần 2 và phần 3.



**Phần luật dịch:** Mỗi luật dịch là một luật sinh kết hợp với hành vi ngữ nghĩa.

Mỗi luật sinh có dạng

<vế trái> → <alt1> | <alt2> | ... <altn>

được mô tả trong Yacc :

```
<vế trái>   : <alt1>    { hành vi ngữ nghĩa 1 }
            | <alt2>    { hành vi ngữ nghĩa 2 }
            ...
            | <altn>    { hành vi ngữ nghĩa n }
            ;
```



Trong luật sinh, các ký tự nằm trong cặp dấu nháy đơn. 'c' là một ký hiệu kết thúc c, một chuỗi chữ cái và chữ số không nằm trong cặp dấu nháy đơn và không được khai báo là token sẽ là ký hiệu chưa kết thúc.

*Hành vi ngữ nghĩa của Yacc* là một chuỗi các lệnh C có dạng:

- $$ Giá trị thuộc tính kết hợp với ký hiệu chưa kết thúc bên vế trái.
- $I Giá trị thuộc tính kết hợp với ký hiệu văn phạm thứ i (kết thúc hoặc chưa) của vế phải.

**Phần thủ tục:** Là các thủ tục viết bằng ngôn ngữ C

Ở đây một bộ phân tích từ vựng yylex( ) sinh ra một cặp gồm token và giá trị thuộc tính kết hợp với nó. Các token được trả về phải được khai báo trong phần khai báo. Giá trị thuộc kết hợp với token giao tiếp với bộ phân tích cú pháp thông qua biến yylval (một biến được định nghĩa bởi yacc)

**Chú ý:** Chúng ta có thể kết hợp Lex và Yacc bằng cách dùng **#include "lex.yy.c"** thay cho thủ tục **yylex( )** trong phần thứ 3.

# BÀI TẬP CHƯƠNG IV

**4.1.** Cho văn phạm G chứa các luật sinh sau:

$$S \rightarrow (\ L) \mid a$$

$$L \rightarrow L\ ,\ S \mid S$$

a) Hãy chỉ ra các thành phần của văn phạm phi ngữ cảnh cho G.

b) Viết văn phạm tương đương sau khi loại bỏ đệ quy trái .

c) Xây dựng bộ phân tích cú pháp dự đoán cho văn phạm.

d) Hãy dùng bộ phân tích cú pháp đã được xây dựng để vẽ cây phân tích cú pháp cho các câu nhập sau:

**i) (a, a)**

**ii) (a, (a, a))**

**iii) (a, (a, a), (a, a)))**

e) Xây dựng dẫn xuất trái, dẫn xuất phải cho từng câu ở phần d

f) Hãy cho biết ngôn ngữ do văn phạm G sinh ra ?



**4.2.** Cho văn phạm G chứa các luật sinh sau:

$$S \rightarrow \mathbf{aSbS} \mid \mathbf{bSaS} \mid \varepsilon$$

a) Chứng minh văn phạm này là mơ hồ bằng cách xây dựng 2 chuỗi dẫn xuất trái khác nhau cho cùng câu nhập **abab**.

b) Xây dựng các chuỗi dẫn xuất phải tương ứng cho câu nhập **abab**.

c) Vẽ các cây phân tích cú pháp tương ứng.

d) Văn phạm này sinh ra ngôn ngữ gì ?

e) Xây dựng bộ phân tích cú pháp đệ quy lùi cho văn phạm trên. Có thể xây dựng bộ phân tích cú pháp dự đoán cho văn phạm này không ?



**4.3.** Cho văn phạm G chứa các luật sinh sau:

bexpr → bexpr **or** bterm | bterm

bterm → bterm **and** bfactor | bfactor

bfactor → **not** bfactor | (bexpr) | **true** | **false**

a) Hãy xây dựng bộ phân tích cú pháp dự đoán cho văn phạm G.

b) Xây dựng cây phân tích cú pháp cho câu : **not ( true and false )**

c) Chứng minh rằng văn phạm này sinh ra toàn bộ các biểu thức boole.

d) Văn phạm G có là văn phạm mơ hồ không ? Tại sao ?

e) Xây dựng bộ phân tích cú pháp SLR cho văn phạm.

**4.4.** Cho văn phạm G chứa các luật sinh sau:

$$R \rightarrow R + R \mid RR \mid R^* \mid (R) \mid \mathbf{a} \mid \mathbf{b}$$

a) Chứng minh rằng văn phạm này sinh ra mọi biểu thức chính quy trên các ký hiệu a và b.

b) Chứng tỏ đây là văn phạm mơ hồ.

c) Xây dựng văn phạm không mơ hồ tương đương với thứ tự ưu tiên của các phép tóan giảm dần như sau : phép bao đóng, phép nối kết, phép hợp.

d) Vẽ cây phân tích cú pháp trong cả hai văn phạm trên cho câu nhập : **a + b * c**

e) Xây dựng bộ phân tích cú pháp dự đoán từ văn phạm không mơ hồ.

f) Xây dựng bảng phân tích cú pháp SLR cho văn phạm G. Đề nghị một quy tắc giải quyết đụng độ sao cho các biểu thức chính quy được phân tích một cách bình thường.

**4.5.** Văn phạm sau đây là một đề nghị điều chỉnh tính mơ hồ cho văn phạm chứa câu lệnh **if - then - else**:



$$\begin{aligned} \textit{Stmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \\ &\mid \text{matched\_stmt} \\ \textit{Matched\_Stmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matched\_stmt}\ \textbf{else}\ \text{stmt} \\ &\mid \textbf{other} \end{aligned}$$

Chứng minh rằng văn phạm này vẫn mơ hồ.

**4.6.** Thiết kế văn phạm cho các ngôn ngữ sau. Ngôn ngữ nào là chính quy?

a) Tập tất cả các chuỗi 0 và 1 sao cho mỗi số 0 có ít nhất một số 1 ở ngay sau nó.

b) Các chuỗi 0 và 1 với số số 0 bằng số số 1.

c) Các chuỗi 0 và 1 với số số 0 không bằng số số 1.



d) Các chuỗi 0 và 1 không chứa chuỗi 001 như chuỗi con.

4.7. Cho văn phạm G chứa các luật sinh sau :

$$S \rightarrow \mathbf{aSa} \mid \mathbf{aa}$$

Xây dựng bộ phân tích cú pháp đệ quy lùi cho văn phạm với yêu cầu phải thử khả triển aSa trước aa.

**4.8. Cho** văn phạm G chứa các luật sinh sau:

S → **a**AB

A → A**bc** | **b**

B → **d**

a) Xây dựng bộ phân tích cú pháp dự đoán cho văn phạm .

b) Hãy dùng bộ phân tích cú pháp đã được xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **abbcd**

**4.9.** Cho văn phạm G chứa các luật sinh sau:

E → E **or** T | T

T → T **and** F | F

F → **(** E**)** | **not** F | **id**

a) Hãy xây dựng bộ phân tích cú pháp dự đoán cho văn phạm.

b) Vẽ cây phân tích cú pháp cho câu nhập : id and not **( id or id )**

**4.10.** Cho văn phạm G chứa các luật sinh sau:



S → AB

A → A**b** | **a**

B → ***c***B | **d**

a) Xây dựng bộ phân tích cú pháp thứ tự ưu tiên cho văn phạm .

b) Hãy dùng bộ phân tích cú pháp đã xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **abccd**

**4.11.** Cho văn phạm G:

S → D • D | D

D → DB | B



B → **0** | **1**

a) Xây dựng bộ phân tích cú pháp thứ tự ưu tiên cho văn phạm .

b) Hãy dùng bộ phân tích cú pháp đã xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **101•101**

**4.12.** Cho văn phạm G

Assign → **id : = exp**

Exp → Exp + Term | Term

Term → Term * Factor | Factor

Factor → **id** | ***(*** Exp ***)***

a) Xây dựng bộ phân tích cú pháp thứ tự ưu tiên cho văn phạm .

b) Hãy dùng bộ phân tích cú pháp đã được xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **id : = id + id * id**

**4.13.** Cho văn phạm mơ hồ như sau:

S → AS | **b**

A → SA | **a**

a) Xây dựng họ tập hợp mục LR(0) cho văn phạm này.

b) Xây dựng bảng phân tích cú pháp SLR .

c) Thực hiện quá trình phân tích cú pháp SLR khả triển cho chuỗi nhập : **abab**

d) Xây dựng bảng phân tích cú pháp chính tắc .

e) Xây dựng bảng phân tích cú pháp LALR .

**4.14.** Cho văn phạm G như sau:



E → E + T | T

T → TF | F

F → F * | **a** | **b**

a) Xây dựng bảng phân tích cú pháp SLR cho văn phạm này.

b) Thực hiện quá trình phân tích cú pháp SLR cho chuỗi nhập : **b + ab* a**

c) Xây dựng bảng phân tích cú pháp LALR.

**4.15.** Chứng tỏ rằng văn phạm sau đây:

S → **Aa** | **bAc** | **dc** | **bda**

A → **d**



là LALR(1) nhưng không phải SLR(1).

**4.16.** Cho văn phạm G như sau:

E → E **sub** R | E **sup** E | { E } | **c**

R → E **sup** E | E

a) Xây dựng bảng phân tích cú pháp SLR cho văn phạm này.

b) Đề nghị một quy tắc giải quyết đụng độ để các biểu thức text có thể được phân tích một cách bình thường.

**4.17.** Viết một chương trình Yacc nhận chuỗi input là các biểu thức số học, sinh ra output là chuỗi biểu thức hậu tố tương ứng.

**4.18.** Viết một chương trình Yacc nhận biểu thức chính quy làm chuỗi input và sinh ra output là cây phân tích cú pháp của nó.