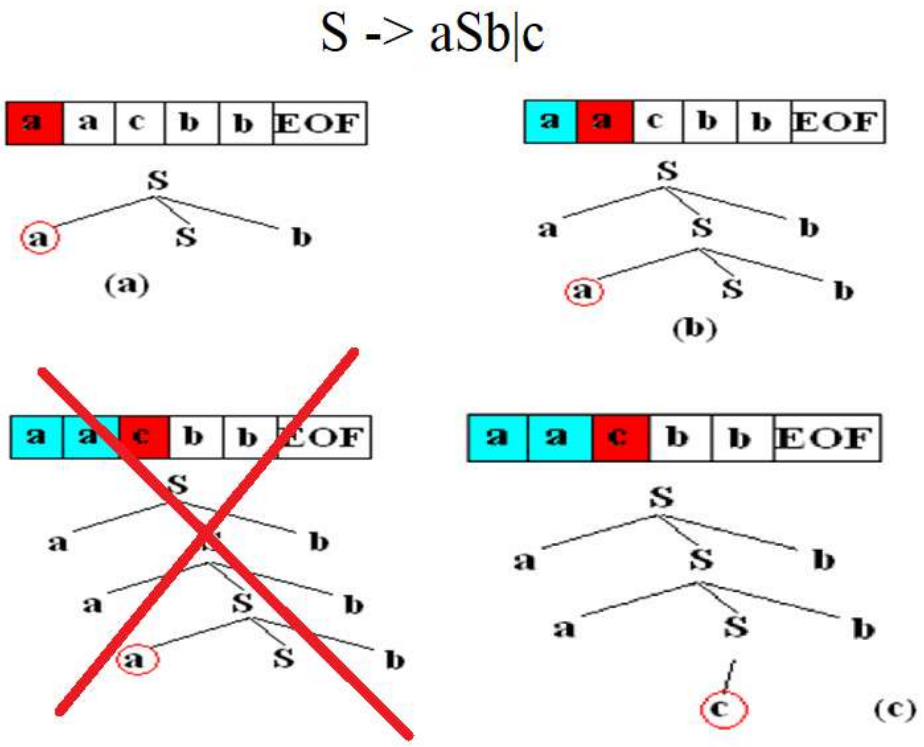
Bài 7
Phân tích cú pháp tiền định

ONE LOVE. ONE FUTURE.

Quay lại giải thuật phân tích cú pháp trên xuống quay lui

- Liệu có thể chọn vế phải của sản xuất với vế trái là nút hoạt động?
- Điều đó phụ thuộc dạng của văn phạm



Phân tích tiền định

- Tư tưởng chính của giải thuật phân tích cú pháp trên xuống **quay lui**
  - Bắt đầu từ gốc, phát triển xuống các nút cấp dưới
  - Chọn một sản xuất và thử xem có phù hợp với xâu vào không
  - Quay lui nếu lựa chọn dẫn đến ký hiệu được sinh bởi văn phạm không phù hợp ký hiệu đang xét
- Có thể tránh được **quay lui**?
  - Cho sản xuất $A \rightarrow \alpha \mid \beta$ bộ phân tích cú pháp cần chọn giữa α và β
- Làm thế nào?
  - Cho ký hiệu không kết thúc A và ký hiệu xem trước t, sản xuất nào của A chắc chắn sinh ra một xâu bắt đầu bởi t?

Bộ phân tích tiền định

- Bộ phân tích cú pháp có thể "đoán trước" sản xuất nào sẽ được dùng khi nút hoạt động là ký hiệu không kết thúc
  - Bằng cách xem trước 1 hoặc nhiều hơn các token
  - Không quay lui
- Bộ phân tích cú pháp tiền định hoạt động trên các văn phạm LL(k)
  - L bên trái là "left-to-right" : hướng đọc file nguồn
  - L bên phải là "leftmost derivation": phân tích trên xuống, trả ra phân tích trái
  - k là số token phải xem trước để "đoán" vế phải được chọn
- Phương pháp phân tích LL(1) được dùng trong thực tế

Bộ phân tích cú pháp tiền định cho văn phạm LL(1)

- Sử dụng *bảng phân tích M* và *stack* (nguyên mẫu của stack D2 trong bộ phân tích trên xuống quay lui).
- Để thực hiện giải thuật này, văn phạm phải thỏa mãn điều kiện
  - *Không có 2 vế phải của cùng một vế trái sinh ra các xâu có tiền tố giống nhau*
- Đặc biệt cũng giống như các phương pháp phân tích trên xuống khác, văn phạm cần không đệ quy trái để tránh lặp vô hạn

Điều kiện bắt buộc

Văn phạm phải không and no two right sides of a production have a common prefix.

It is obvious if the grammar is LL(1)

Đệ quy trái

Văn phạm G là *đệ quy trái* nếu nó chứa một ký hiệu không kết thúc A với suy dẫn:

$$A \Rightarrow^* A\alpha \qquad \text{với } \alpha \text{ là xâu bất kỳ}$$

Đệ quy trái có thể xuất hiện trong một bước suy dẫn (*đệ quy trái trực tiếp*) hoặc nhiều bước suy dẫn (*đệ quy trái gián tiếp*)

Tuy nhiên, ta đã chứng minh được là có thể khử đệ quy trái trong văn phạm bằng cách thay thế các sản xuất đệ quy trái bằng các sản xuất tương đương không đệ quy trái.

Đệ quy trái trực tiếp

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

Left recursion - problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive,

but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate left recursion - Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- **for** i **from** 1 **to** n **do** {
  - **for** j **from** 1 **to** i-1 **do** {

    replace each production

    $A_i \rightarrow A_j \gamma$

    by

    $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

    where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

  }
  - eliminate immediate left-recursions among A_i productions

}

Example of removing immediate left recursion

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

$F \rightarrow id \mid (E)$

Predictive Parsing and Left Factoring

- In the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow int \mid int * T \mid (E)$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- A grammar must be left-factored before use for predictive parsing

Left Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

  if_stmt → if expr then stmt else stmt |
  if expr then stmt

- When we see nested if, we cannot know which production rule to choose to re-write *stmt* in the derivation.

Left Factoring (con'd)

In general,

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

when processing α we cannot know whether expand

A to $\alpha\beta_1$ or

A to $\alpha\beta_2$

But, if we re-write the grammar as follows

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left factoring algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left Factoring example

S → if E then S | if E then S else S

can be rewritten as

S → if E then S S'

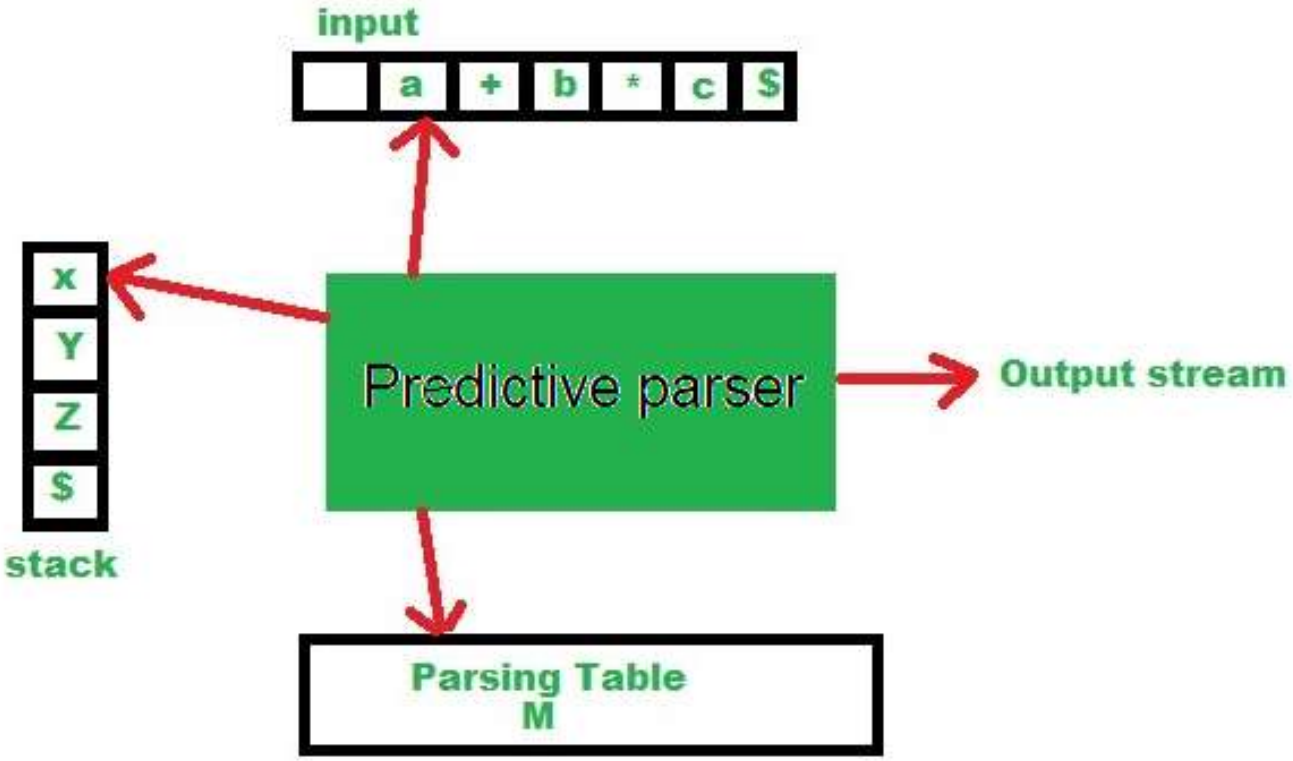
S' → else S | ε

In KPL

```
IfSt ::= KW_IF Condition KW_THEN Statement ElseSt

ElseSt ::= KW_ELSE Statement
ElseSt ::= ε
```

Predictive parser



Parsing table M

- *M*[*X*, *token*] indicates which production to use if the top of the stack is a nonterminal *X* and the current token is equal to *token*; F
- in that case we pop *X* from the stack and we push all the rhs symbols of the production *M*[*X*, *token*] .
- We use a special symbol $ to denote the end of file. Let *S* be the start symbol

Predictive Parser

The input contains the string to be parsed, followed by $ (EOF)

The stack contains a sequence of grammar symbols, preceded by #, the bottom-of-stack marker.

Initially the stack contains the start symbol of the grammar preceded by $.

The parsing table is a two dimensional array M[A,a], where A is a nonterminal, and a is a terminal or the symbol $.

- The parser is controlled by a program that behaves as follows:
- The program determines *X*, the symbol on top of the stack, and a, the current input symbol.
- These two symbols determine the action of the parser.

There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops *X* off the stack and advances the input pointer to the next input symbol.
3. If *X* is a nonterminal, the program consults entry M[*X*,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry.

If M[*X*,a] = {$X \rightarrow UVW$}, the parser replaces *X* on top of the stack by *WVU* (with *U* on top).

If M[*X*,a] = error, the parser calls an error recovery routine.

Parsing table for grammar S→ aSb | c

	a	b	c	$
S	S → aSb	Error	S → c	Error
a	Push	Error	Error	Error
b	Error	Push	Error	Error
c	Error	Error	Push	Error
#	Error	Error	Error	Accept

LL(1) Parsing Tables. Errors

- Yellow entries indicate error situations
  - Consider the [S,b] entry
  - “There is no way to derive a string starting with b from non-terminal S

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S #>
repeat
   case stack of
      <T, rest>  : if M[X,*next] = T → Y1…Yn
                              then stack ← <Y1… Yn rest>;
                              else  error ();   //X-nonterminal T
      <t, rest>   : if t == *next ++
                              then  stack ← <rest>;
                              else error (); //X-nonterminal
until stack == < >
//*next refers to the symbol to be checked
```

LL(1) Parsing Example for aacbb

Stack	Input	Action
S#	aacbb$	$S \rightarrow aSb$
aSb#	aacbb$	push
Sb#	acbb$	$S \rightarrow aSb$
aSbb#	acbb$	push
Sbb#	cbb$	$S \rightarrow c$
cbb#	cbb$	push
bb#	bb$	push
b#	b$	push
#	$	ACCEPT

Phân tích tiền định

- Nếu có hai sản xuất: $A \rightarrow \alpha \mid \beta$, ta mong muốn có một phương pháp rõ ràng để chọn đúng sản xuất cần thiết
- Định nghĩa:
  - Với α là một xâu chứa ký hiệu kết thúc và không kết thúc, **x** $\in$ FIRST(α) nếu từ α có thể suy dẫn ra **x**γ (x chứa 0 hoặc 1 ký hiệu)
- Nếu FIRST(α) và FIRST(β) không chứa ký hiệu chung ta biết phải chọn $A \rightarrow \alpha$ hay $A \rightarrow \beta$ khi đã xem trước một ký hiệu

Phân tích tiền định

- Tính FIRST(X):
  - Nếu X là ký hiệu kết thúc FIRST(X)={X}
  - Nếu $X \rightarrow \varepsilon$ là một sản xuất thì thêm ε vào FIRST(X)
  - Nếu X là ký hiệu không kết thúc và $X \rightarrow Y_1 Y_2 \ldots Y_n$ là một sản xuất ,
    - Thêm FIRST(Y_1) vào FIRST(X)
    - Thêm FIRST(Y_{i+1}) vào FIRST(X) nếu FIRST(Y_1),. . . FIRST(Y_j) chứa ε
- Tính FIRST(α) tương tự bước thứ ba trong tính FIRST(X)

Phân tích tiền định

- Nếu ta có sản xuất để chọn là $A \rightarrow \alpha$ với $\alpha = \varepsilon$ hoặc $\alpha \Rightarrow^* \varepsilon$? Ký hiệu nào sẽ là ký hiệu đầu tiên được sinh bởi một dạng câu chứa A?
- Có thể mở rộng A nếu ta biết rằng **tồn tại một dạng câu mà ký hiệu đang xét xuất hiện sau A**, nghĩa là ký hiệu đang xét thuộc FOLLOW(A)
- Định nghĩa:
  - Với A là ký hiệu không kết thúc, $\mathbf{x} \in$ FOLLOW(A) nếu và chỉ nếu S có thể suy dẫn ra $\alpha A \mathbf{x} \beta$, $|x| = 1$ hoặc $x = \varepsilon$ (khi ấy β cũng là ε)

Tính FOLLOW

- FOLLOW(S) chứa ε (EOF)
- Với các sản xuất dạng $A \rightarrow \alpha B \beta$, mọi ký hiệu trong FIRST(β) trừ ε tham gia vào FOLLOW(B)
- Với các sản xuất dạng $A \rightarrow \alpha B$ hoặc $A \rightarrow \alpha B \beta$ trong đó FIRST(β) chứa ε, FOLLOW(B) chứa mọi ký hiệu của FOLLOW(A) và ε (hoặc $)

Phân tích tiền định

- Với các khái niệm
  - FIRST
  - FOLLOW
- Ta có thể xây dựng bộ phân tích cú pháp mà không đòi hỏi quay lui
- Chỉ có thể xây dựng bộ phân tích cú pháp như vậy cho những văn phạm đặc biệt
- Loại văn phạm như vậy bao gồm văn phạm một số ngôn ngữ lập trình đơn giản, chẳng hạn KPL,PL/0, PASCAL-S

Bảng phân tích tiền định

- Dùng cho bộ sinh phân tích cú pháp
- Đầu vào của giải thuật: văn phạm G và xâu w
- Căn cứ
  - Ký hiệu đang xét
  - Ký hiệu đang ở đỉnh stack
- Quyết định
  - Thay thế ký hiệu không kết thúc
  - Chuyển con trỏ sang ký hiệu tiếp
  - Chấp nhận xâu
  - Thông báo lỗi

Bộ phân tích cú pháp tiền định

Vào: Văn phạm phi ngữ cảnh LL(1) G
Xâu w

Các thành phần cơ bản

- Stack
- Bảng phân tích
- Băng vào
- Chương trình phân tích

Mô tả các thành phần

- Băng vào chứa xâu cần phân tích, kết thúc bằng $ (EOF)
- Stack giống như stack D2 của bộ phân tích cú pháp top down quay lui, # ở đáy của stack. Ban đầu S ở đỉnh stack, trên ký hiệu #.
- Bảng phân tích M[A,a] với A là một ký hiệu của văn phạm, a là ký hiệu kết thúc hoặc $.

Hoạt động của bộ phân tích cú pháp

- Nếu stack còn lại # (đáy), đầu đọc chỉ $ (EOF), dừng và đoán nhận xâu.
- If X=a (ký hiệu kết thúc đang xét trên xâu vào) và không là $, xóa X trên đỉnh stack , chuyển đầu đọc sang ô kế tiếp.
- Nếu X là ký hiệu không kết thúc, bộ PTCP tra bảng phân tích cú pháp M, tìm ô M[X,a], thay thế ký hiệu đỉnh stack (X) bằng vế phải sản xuất trong ô (nếu có). Nếu là ô rỗng -> ERROR, gọi thủ tục thông báo lỗi.

Bảng phân tích LL(1)

- Dùng cho bộ sinh phân tích cú pháp
- Căn cứ
  - Ký hiệu đang xét
  - Ký hiệu đang ở đỉnh stack
- Quyết định
  - Thay thế ký hiệu không kết thúc
  - Chuyển con trỏ sang ký hiệu tiếp
  - Chấp nhận xâu

Giải thuật xây dựng bảng phân tích

1. Với mỗi sản xuất $A \rightarrow \alpha$ của văn phạm G, thực hiện các bước 2 và 3.
2. Với mỗi ký hiệu kết thúc $a \in FIRST(\alpha)$, thêm $A \rightarrow \alpha$ vào M[A,a].
3. If ε thuộc $FIRST(\alpha)$, thêm $A \rightarrow \alpha$ vào M[A,b] với mỗi b thuộc FOLLOW(A). If ε thuộc $FIRST(\alpha)$, và $ thuộc FOLLOW(A), thì thêm $A \rightarrow \alpha$ vào M[A,$]
4. Các ô M(a,a) với a là ký hiệu kết thúc, thêm hành động "đẩy"
5. M[#,$] ="nhận"
6. Các ô còn lại đánh dấu là "lỗi".

Ví dụ

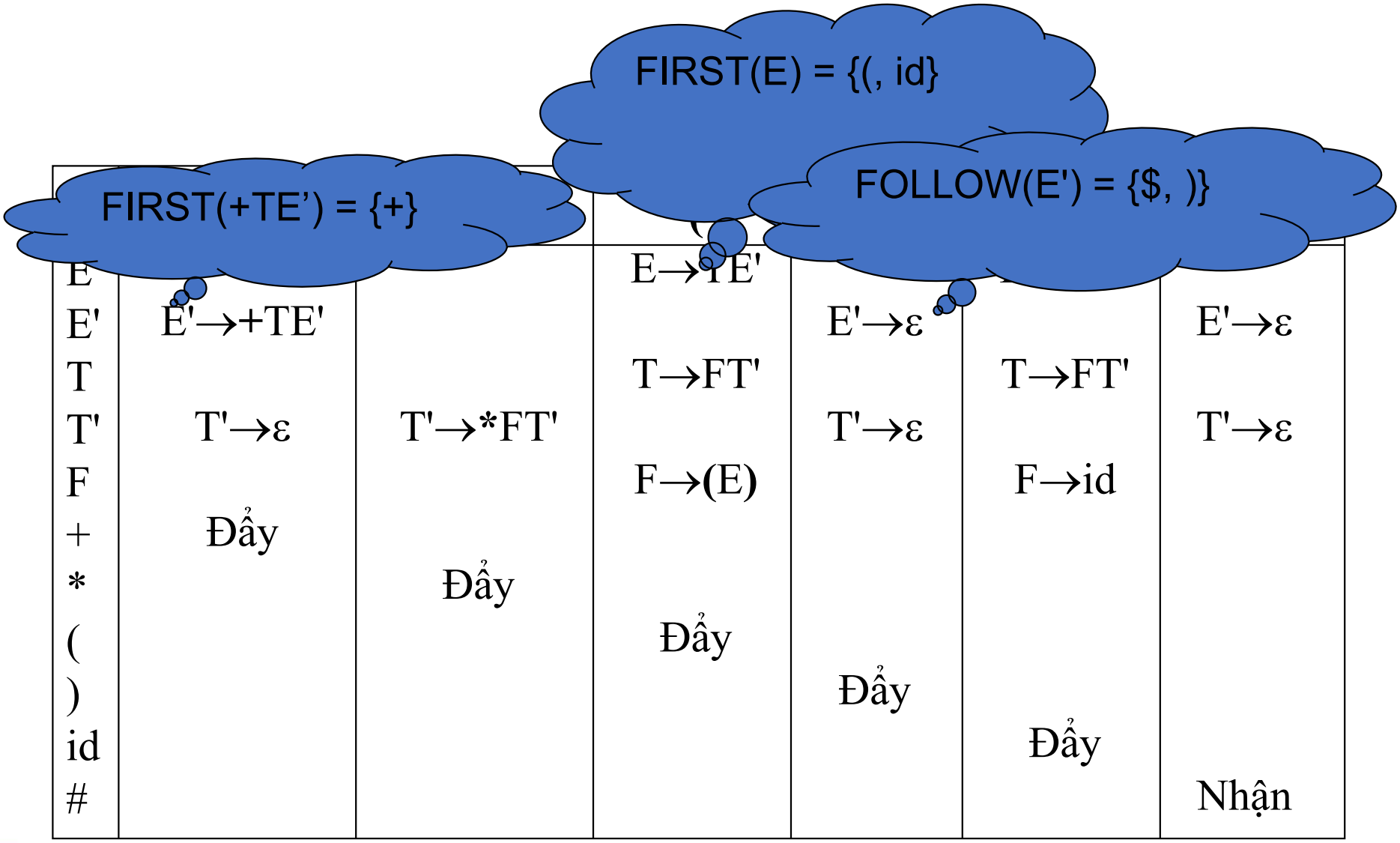
- Văn phạm:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

FIRST(+TE') = {+}
FOLLOW(E') = {$,)}
FIRST(*FT') = {*}
FOLLOW(T') = {+, $,)}
FIRST((E)) = {(}
FIRST(id) = {id}

Văn phạm này LL(1)
có thể xây dựng bộ phân tích tiền định

Bảng phân tích



E			E→TE'			
E'	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→id	
+	Đẩy					
*		Đẩy				
(			Đẩy			
)				Đẩy		
id					Đẩy	
#						Nhận

Phân tích xâu vào id*id sử dụng bảng phân tích và stack

Bước	Stack	Xâu vào	Hành động kế tiếp
1	#E	id*id$	E→TE'
2	#E'T	id*id$	T→FT'
3	#E'T'F	id*id$	F→id
4	#E'T'id	id*id$	đẩy id
5	#E'T'	*id$	T'→*FT'
6	#E'T'F*	*id$	đẩy *
7	#E'T'F	id$	F→id
8	#E'T'id	id$	đẩy id
9	#E'T'	$	T'→ε
10	#E'	$	E' →ε
11	#	$	nhận