



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

# NGÔN NGỮ VÀ PHƯƠNG PHÁP DỊCH

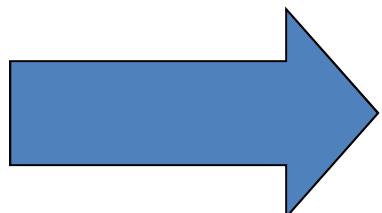
TS. Nguyễn Thị Thu Hương –  
Bộ môn Khoa học máy tính  
Viện CNTT – TT – ĐHBKHN  
Tel (04) 38696121 - Mobi : 0903253796  
Email :[huongnt@soict.hust.edu.vn](mailto:huongnt@soict.hust.edu.vn),  
[huong.nguyenthithu@hust.edu.vn](mailto:huong.nguyenthithu@hust.edu.vn)

# Môn học sẽ nghiên cứu

- Cách thức làm việc của bộ xử lý ngôn ngữ và chương trình dịch
- Sinh mã trung gian và mã máy cho những cấu trúc ngôn ngữ cụ thể
- Thiết kế ngôn ngữ: cú pháp và ngữ nghĩa
- Cách thức xây dựng một chương trình dịch đơn giản
- Cách thức sử dụng các bộ sinh chương trình dịch
- Cách thức làm việc của máy tính (tập lệnh, thanh ghi, các cấu trúc dữ liệu được sử dụng khi thực hiện. . . )

# Tại sao cần nghiên cứu CT dịch?

- Rèn kỹ năng phát triển ứng dụng quy mô lớn
- Làm việc với các cấu trúc dữ liệu phức tạp
- Tìm hiểu sự tương tác giữa các giải thuật



Bước chuẩn bị cho những  
dự án lớn trong tương lai.

# Những vấn đề chính

- Bộ xử lý ngôn ngữ
- Cấu trúc của một trình biên dịch (1 pha)
- Công cụ biểu diễn cú pháp
- Phân tích từ vựng
- Phương pháp chung để phân tích cú pháp
- Phân tích cú pháp tiền định
- Bảng ký hiệu
- Phân tích ngũ nghĩa
- Sinh mã trung gian
- Tối ưu mã
- Sinh mã đích
- Bộ sinh Compiler

# Tài liệu tham khảo

- Aho.A.V, Sethi.R., Lam M., Ullman.J.D.  
*Compiler : Principles, Techniques and Tools.*  
Addison Wesley. 2007.
- Andrew.W.Appel  
Modern Compiler Implementation in Java (C)  
*Princeton University. 2002*
- Nguyễn Văn Ba  
*Giáo trình kỹ thuật biên dịch*  
Đại học Bách Khoa Hà Nội.1994
- Niklaus Wirth  
*Compiler Construction.*  
Addison Westley. 1996 (2014)
- Bài giảng về ngôn ngữ và phương pháp dịch
- [www.sourceforge.net](http://www.sourceforge.net)

# Tài liệu tham khảo

- Bal.H. E.  
*Modern Compiler Design.*  
John Wiley & Sons Inc (2000)
- William Allan Wulf.  
*The Design of an Optimizing Compiler*  
Elsevier Science Ltd (1980)
- Charles N. Fischer.  
*Crafting a Compiler*  
Benjamin-Cummings Pub Co (1987)

# Đánh giá kết quả học tập

- Giữa kỳ (30%): Bài tập (80%) + chuyên cần (20%)
- Cuối kỳ (70%): Thi trắc nghiệm

# Nội dung học tập

- Lý thuyết: Nguyên lý xây dựng chương trình dịch
- Bài tập: nghiên cứu và hoàn thiện một chương trình dịch đơn giản cho ngôn ngữ KPL (Kyoto Programming Language)

# Chương 1: Khái niệm cơ bản

- 1.1. Đặc trưng của ngôn ngữ lập trình
- 1.2. Bộ xử lý ngôn ngữ (Language Processor)
- 1.3. Cấu trúc của chương trình dịch
- 1.4. Văn phạm và ngôn ngữ

## 1.1. Đặc trưng của ngôn ngữ lập trình

- Các ngôn ngữ lập trình được chia thành 5 thế hệ.
- Việc phân chia cấp cao hay thấp phụ thuộc mức độ trừu tượng của ngôn ngữ
  - Cấp thấp : gần với máy
  - Cấp cao : gần với ngôn ngữ tự nhiên

# Thế hệ thứ nhất và thứ hai

- Thế hệ thứ nhất : ngôn ngữ máy
- Thế hệ thứ hai : Assembly
- Các ngôn ngữ thuộc thế hệ thứ nhất và thứ hai là ngôn ngữ lập trình cấp thấp

## Thế hệ thứ ba

- Dễ hiểu hơn
- Cho phép thực hiện các khai báo, chẳng hạn biến
- Phần lớn các ngôn ngữ cho phép lập trình cấu trúc
- Ví dụ: Fortran, Cobol, C, C++, Basic . . .

# Thế hệ thứ tư

- Thường được sử dụng trong một lĩnh vực cụ thể (chẳng hạn thương mại)
- Dễ lập trình, xây dựng phần mềm
- Có thể kèm công cụ tạo form, báo cáo
- Ví dụ :SQL, Visual Basic, Oracle (SQL plus, Oracle Form, Oracle Report). . . .

## Thế hệ thứ năm

- Giải quyết bài toán dựa trên các ràng buộc đưa ra cho chương trình chứ không phải giải thuật của người lập trình.
- Việc giải quyết bài toán do máy tính thực hiện
- Phần lớn các ngôn ngữ dùng để lập trình logic, giải quyết các bài toán trong lĩnh vực trí tuệ nhân tạo

# Đặc trưng của ngôn ngữ lập trình cấp cao

- Độc lập với máy tính
- Gần với ngôn ngữ tự nhiên
- Chương trình dễ đọc, viết và bảo trì
- Muốn thực hiện chương trình phải dịch sang ngôn ngữ máy
- Chương trình thực hiện chậm hơn

## Cú pháp và ngũ nghĩa của ngôn ngữ lập trình

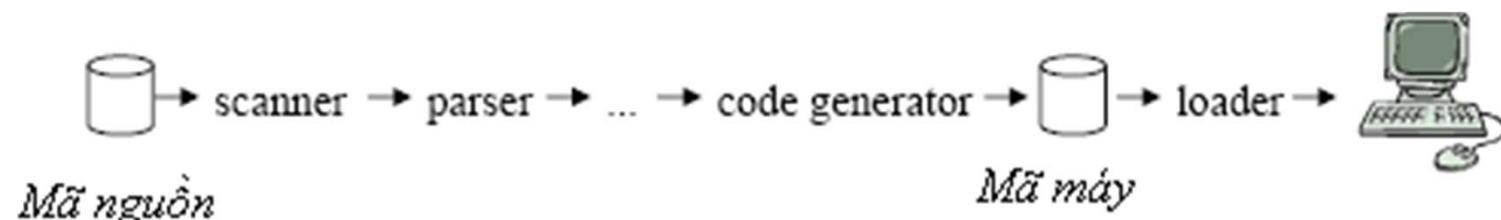
- Cú pháp : Chính tả và văn phạm của các cấu trúc ngôn ngữ
- Ngữ nghĩa : Ý nghĩa và hiệu quả của các cấu trúc ngôn ngữ

## 1.2. Bộ xử lý ngôn ngữ

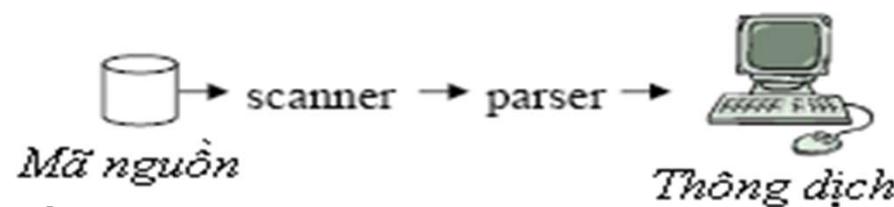
- Phần mềm dịch từ một ngôn ngữ nào đó sang mã máy (có thể đồng thời thực thi)
- Ví dụ
  - Compiler
  - Assembler
  - Interpreter
  - Compiler - Compiler

# Compiler & Interpreter

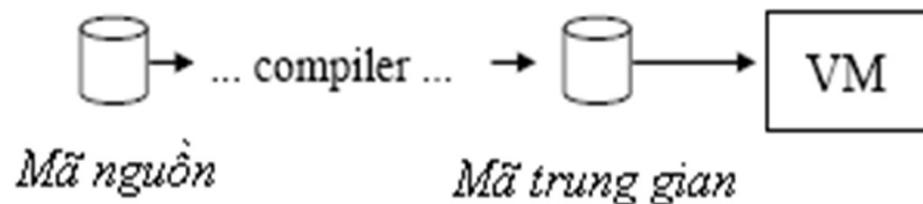
- Compiler : Dịch trực tiếp ra mã máy



- Interpreter : Trực tiếp thực hiện từng lệnh mã nguồn



- Biến thể của Interpreter : thông dịch mã trung gian



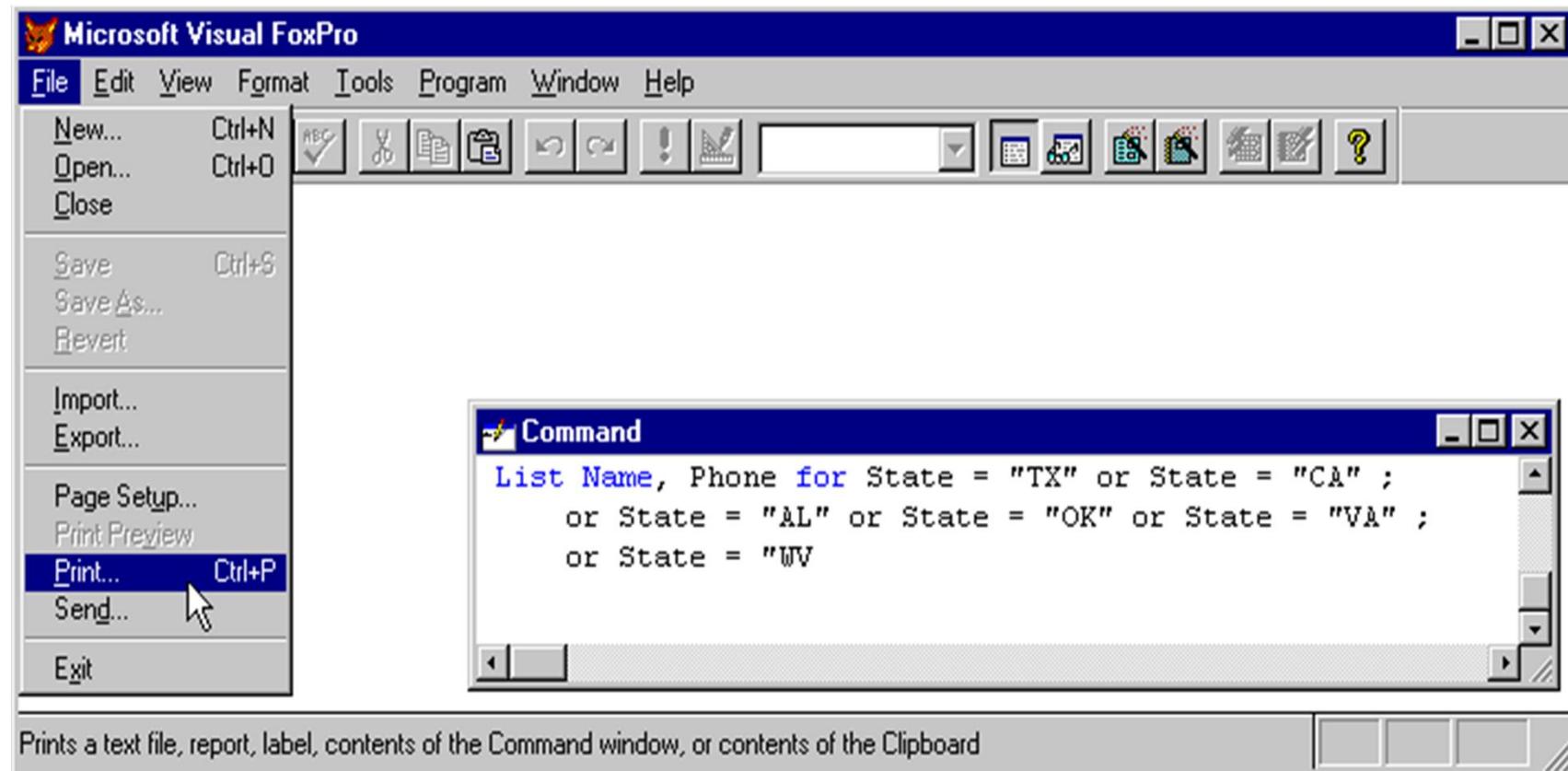
# Interpreter: Trình thông dịch

- Một số ngôn ngữ sử dụng trình thông dịch cho phép dịch và chạy trực tiếp từng lệnh
- Mỗi lệnh được dịch thành một đoạn chương trình trong một ngôn ngữ trung gian. Ngôn ngữ trung gian dùng trình dịch compiler.
- Ngôn ngữ hoàn toàn dùng trình thông dịch: Foxpro
- Ngôn ngữ kết hợp thông dịch và biên dịch: Visual Basic, Python

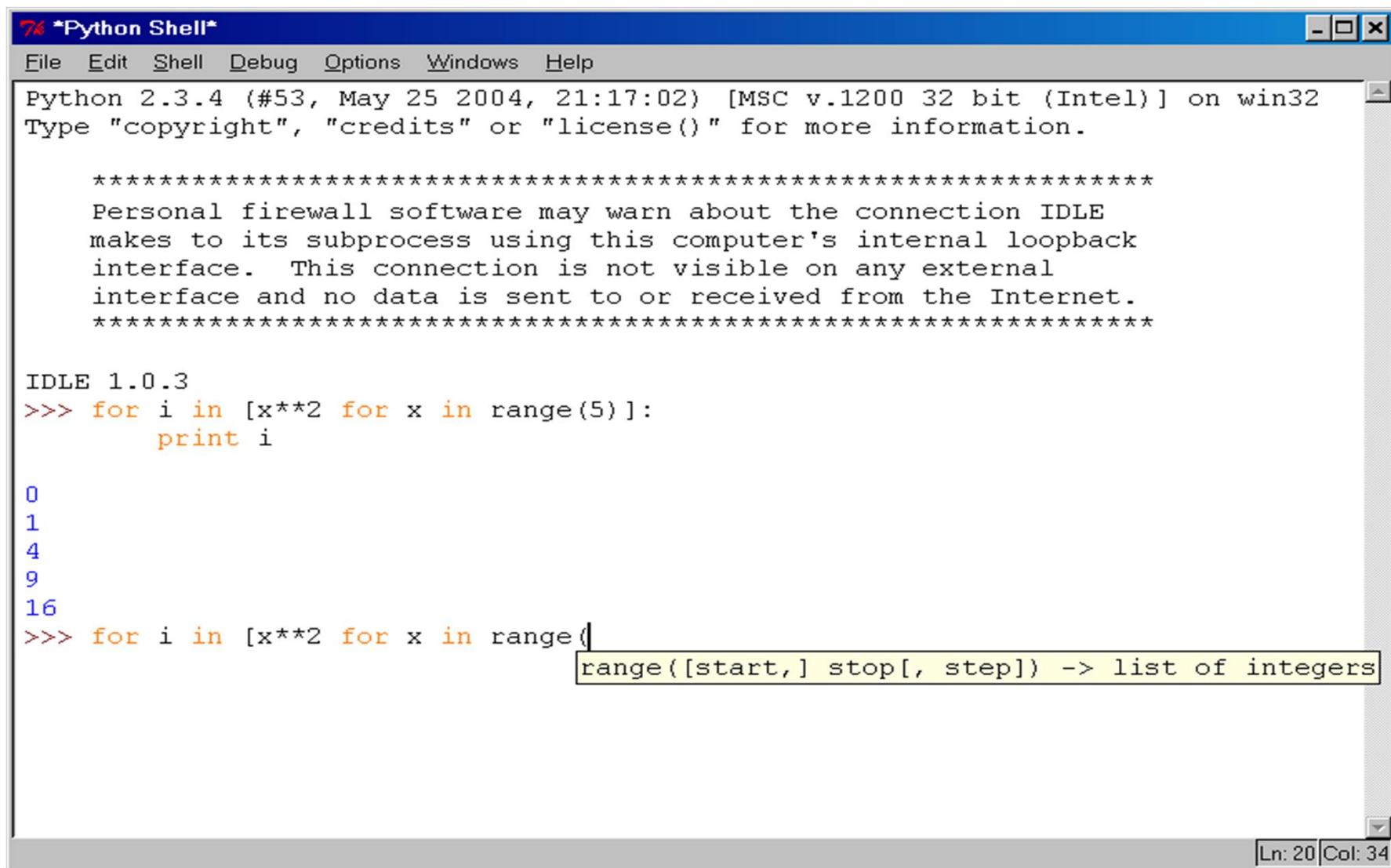
# Ngôn ngữ dùng interpreter: Foxpro

- Hai chế độ làm việc
  - Cửa sổ lệnh: Thực hiện từng lệnh
  - Chương trình: Chạy từng lệnh. Các lệnh trước lệnh đầu tiên bị lỗi trong chương trình vẫn được thực hiện

# Cửa sổ lệnh của Foxpro



# Thực hiện từng lệnh trên Python



The screenshot shows a Python Shell window with the following content:

```
76 *Python Shell*
File Edit Shell Debug Options Windows Help
Python 2.3.4 (#53, May 25 2004, 21:17:02) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****
```

IDLE 1.0.3

```
>>> for i in [x**2 for x in range(5)]:
    print i
```

```
0
1
4
9
16
>>> for i in [x**2 for x in range(1
    range([start,] stop[, step]) -> list of integers
```

Ln: 20 Col: 34

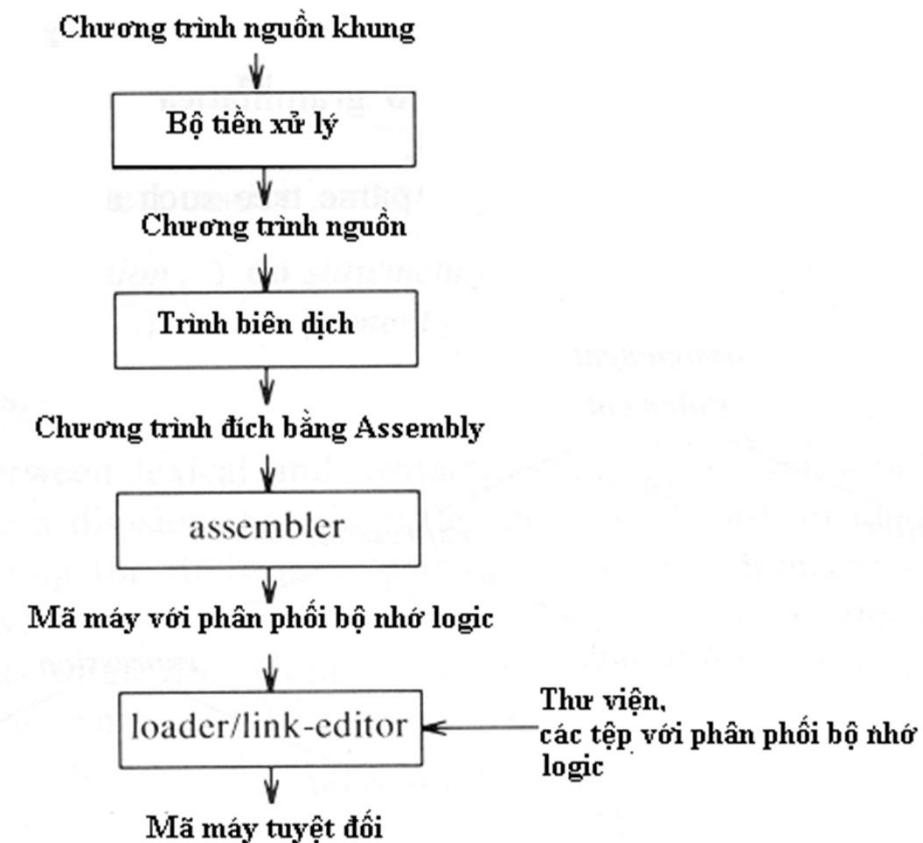
# Compiler (trình biên dịch)

- Mục đích : Dịch chương trình từ ngôn ngữ cấp cao (ngôn ngữ nguồn) sang ngôn ngữ cấp thấp (ngôn ngữ đích).
- Chương trình chỉ thực hiện được khi toàn bộ các lệnh đúng cú pháp.
- Bản thân compiler được viết trên một ngôn ngữ gọi là ngôn ngữ thực hiện

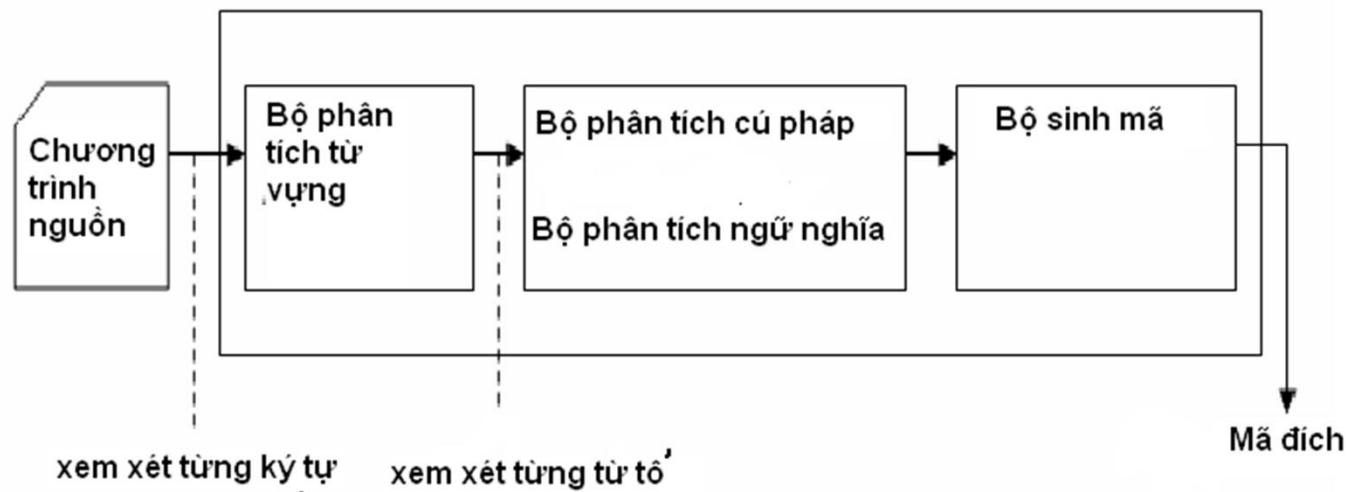
# Các công cụ liên quan đến trình biên dịch

- Trình thông dịch (Interpreter)
- Assembler
- Linker
- Loader
- Bộ tiền xử lý (Preprocessor)
- Editor
- Debugger
- Profiler

# Vị trí của trình biên dịch trong bộ xử lý ngôn ngữ



## 1.3. Cấu trúc của chương trình dịch



# Các giai đoạn của trình biên dịch

- Phân tích từ vựng (Lexical Analysis - Scanner)

Lần lượt xem xét từng ký tự của chương trình nguồn, phân nhóm chúng thành những đơn vị cú pháp gọi là từ tố (token)

- Phân tích cú pháp (Syntax Analysis)

Dãy token do bộ phân tích từ vựng đưa ra được kiểm tra xem có đúng cú pháp không?

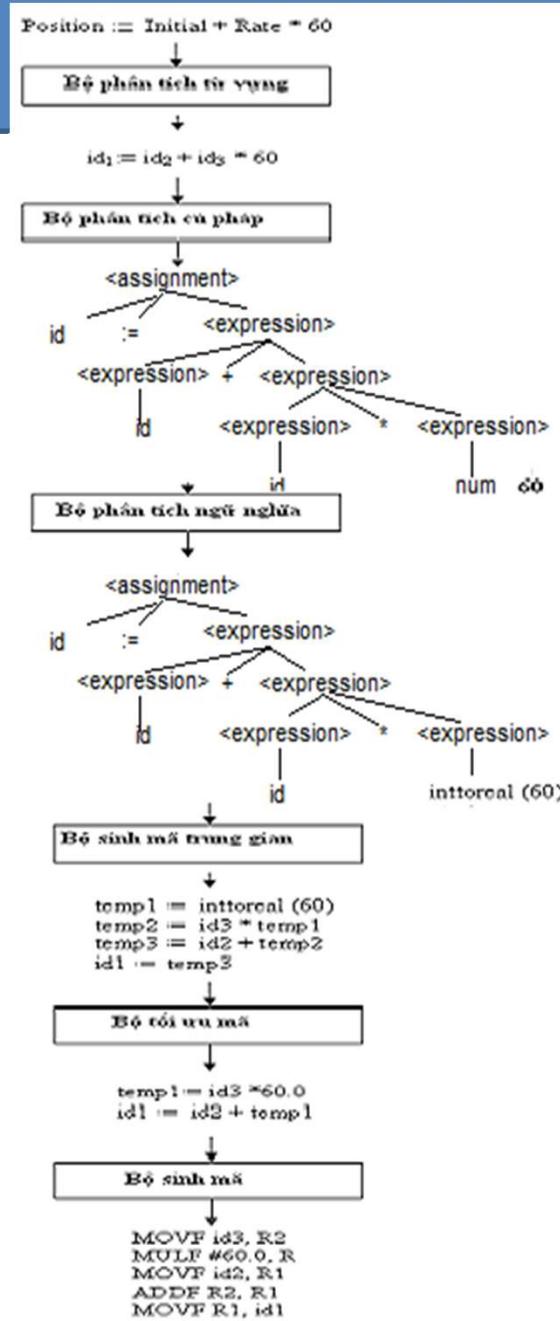
# Các giai đoạn của trình biên dịch

- Phân tích ngữ nghĩa (Semantic Analysis) phân tích ý nghĩa từng lệnh của ngôn ngữ nguồn.
- Sinh mã trung gian (Intermediate Code Generation) thường là mã 3 địa chỉ. Mã trung gian không phụ thuộc máy nên dễ tối ưu.

# Các giai đoạn của trình biên dịch

- Sinh mã đích: Sinh ra các lệnh máy để thực hiện thao tác.
- Tối ưu mã: Thực hiện với mã trung gian và cả mã đích nhằm làm cho chương trình hiệu quả hơn.

# Quá trình dịch một câu lệnh



# Giai đoạn 1: Phân tích từ vựng

- Bộ từ vựng: Chương trình làm nhiệm vụ phân tích từ vựng
- Các công việc của bộ từ vựng
  - Nhóm các ký tự thành từ tố
    - Từ tố :đơn vị cú pháp được xử lý trong quá trình dịch như một thực thể không thể chia nhỏ hơn nữa
    - Nhóm các từ tố theo loại.

# Một số loại từ tố

## LOẠI TỪ TỐ

**identifier**

**number**

=

+

-

;

==

**if**

**else**

(

)

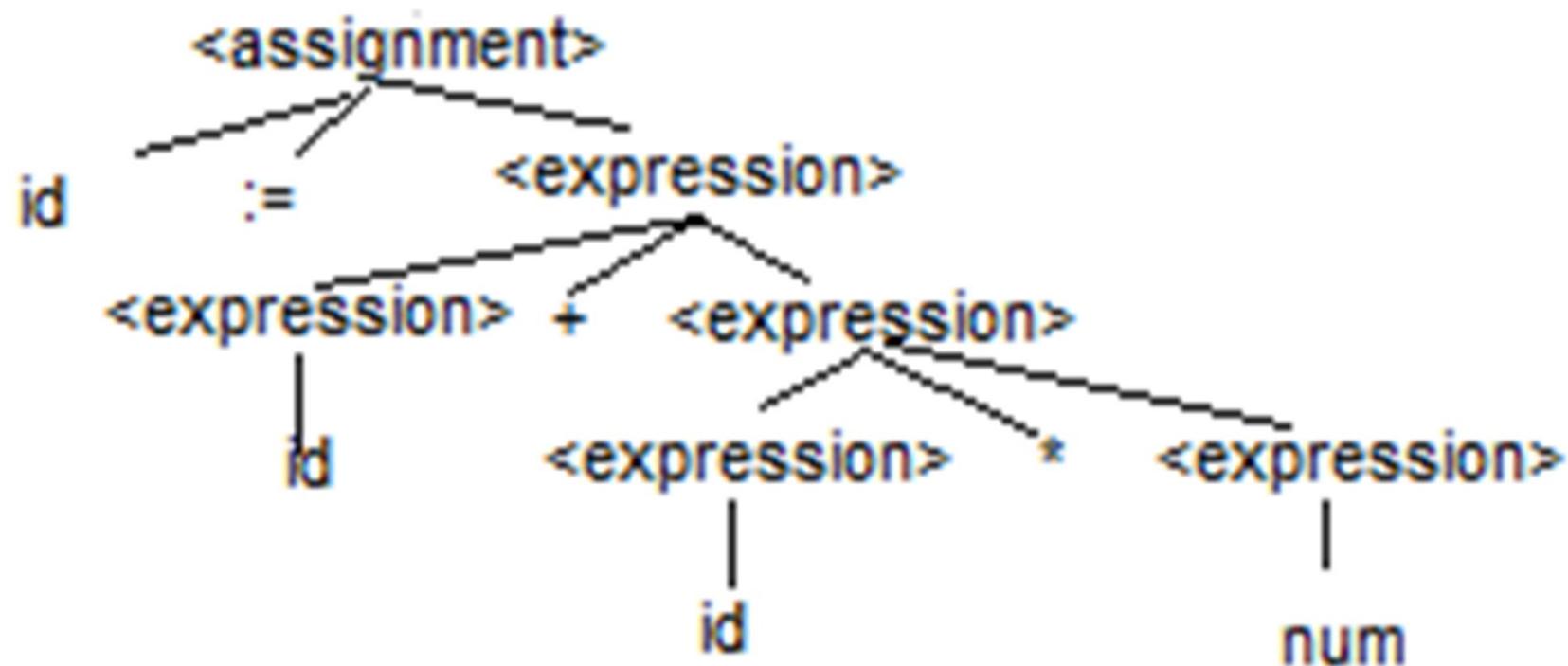
## Giai đoạn 2: Phân tích cú pháp

- Trình biên dịch kiểm tra xem những từ tố mà bộ từ vựng nhận biết được có kết hợp thành những câu lệnh đúng cú pháp không
- Do bộ phân tích cú pháp đảm nhận

## Giai đoạn 2: Phân tích cú pháp

- Đầu ra của bộ phân tích cú pháp:
  - Cây phân tích cú pháp (nếu có)
  - Thông báo lỗi nếu ngược lại
- Việc xây dựng được cây phân tích cú pháp chứng tỏ chương trình đúng về cú pháp

## Cây phân tích cú pháp của câu lệnh c := a+b\*60



# Biểu diễn cú pháp

- Cú pháp
  - Cấu trúc văn phạm của một ngôn ngữ
- Bộ phân tích cú pháp cần đưa ra phân tích cho mỗi câu của ngôn ngữ (chương trình)
- BNF : Dạng chuẩn để mô tả văn phạm của ngôn ngữ
- Sơ đồ cú pháp:cách mô tả văn phạm trực quan dưới dạng đồ thị định hướng

# Khái niệm và kỹ thuật phân tích cú pháp

- Bằng cách áp dụng liên tục các luật mô tả văn phạm hoặc sơ đồ cú pháp
- Nếu bộ PTCP xây dựng được cây phân tích cú pháp thì chương trình là đúng cú pháp. Ngược lại, chương trình không đúng cú pháp

## Khái niệm và kỹ thuật phân tích cú pháp

- Vấn đề quan trọng nhất khi xây dựng trình biên dịch là xây dựng một văn phạm
- Bao gồm đầy đủ các cấu trúc của một chương trình
- Không thể tạo nên một luật nào khác
- Văn phạm phải không nhập nhằng
- Nếu văn phạm nhập nhằng, xây dựng được nhiều hơn 1 cây cho mỗi chương trình được đưa ra phân tích

## Giai đoạn 3: Phân tích ngữ nghĩa

- Duyệt cây cú pháp của chương trình để xem mọi cấu trúc ngữ nghĩa có đúng không
- Chương trình đúng cả về cú pháp và ngữ nghĩa mới sinh mã được

## Giai đoạn 4: Sinh mã trung gian

- Chương trình với mã nguồn được chuyển sang chương trình tương đương trong ngôn ngữ trung gian bằng bộ sinh mã trung gian.
- *Mã trung gian* là mã máy độc lập tương tự với tập lệnh trong máy.

## Ưu điểm của mã trung gian

- 1.Thuận lợi khi cần thay đổi cách biểu diễn chương trình đích.
- 2.Có thể tối ưu hóa mã độc lập với máy đích cho dạng biểu diễn trung gian.
- 3.Giảm thời gian thực thi chương trình đích vì mã trung gian có thể được tối ưu

# Ngôn ngữ trung gian

- Được người thiết kế trình biên dịch quyết định, có thể là:
  - Cây cú pháp
  - Ký pháp Ba Lan sau (hậu tố)
  - Mã 3 địa chỉ ...

## Giai đoạn 5: Sinh mã đích

- Vào: biểu diễn trung gian của chương trình nguồn
- Ra: chương trình đích
  - Mã Assembly
  - Mã mô phỏng trên máy đích ảo

# Các vấn đề thiết kế bộ sinh mã đích

- Input
- Output
- Lựa chọn câu lệnh
- Cấp phát thanh ghi
- Máy đích

## 1.3. Văn phạm và ngôn ngữ

- Trong lý thuyết ngôn ngữ, xâu có thể đóng vai trò của một câu, một chương trình máy tính, một phân tử ADN
- Văn phạm phi ngữ cảnh có thể dùng để sản sinh ra các xâu thuộc ngôn ngữ như sau:

*X = Ký hiệu đầu*

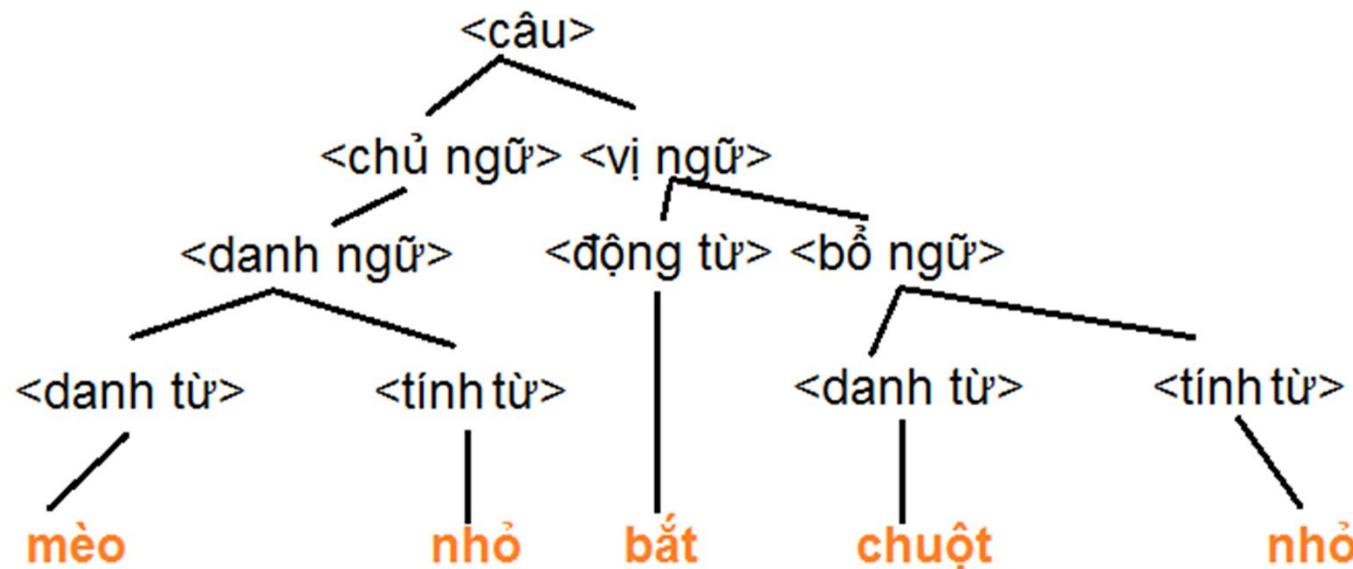
*While còn ký hiệu không kết thúc Y trong X do*

*Áp dụng một trong các sản xuất của văn phạm  
chẳng hạn Y -> w*

Khi X chỉ chứa ký hiệu kết thúc, nó là xâu được sản sinh bởi văn phạm.

# Một văn phạm nhó của ngôn ngữ tự nhiên

<câu> ::= <chủ ngữ> <vị ngữ>  
<chủ ngữ> ::= <danh ngữ>  
<danh ngữ> ::= <danh từ> <tính từ>  
<vị ngữ> ::= <động từ> <bồ ngữ>  
<bồ ngữ> ::= <danh từ> <tính từ>  
<động từ> ::= **bắt**  
<danh từ> ::= **mèo** | **chuột**  
<tính từ> ::= **nhỏ**



# Văn phạm sản sinh các số thực

$<\text{Số}> ::= -<\text{Số thập phân}> | <\text{Số thập phân}>$   
 $<\text{Số thập phân}> ::= <\text{Dãy chữ số}> | <\text{Dãy chữ số}> . <\text{Dãy chữ số}>$   
 $<\text{Dãy chữ số}> ::= <\text{Chữ số}> | <\text{Chữ số}> <\text{Dãy chữ số}>$   
 $<\text{Chữ số}> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

## Định nghĩa hình thức của văn phạm

Văn phạm là bộ 4  $G = (\Sigma, \Delta, P, S)$  , trong đó

$\Sigma$  : Tập hữu hạn các ký hiệu kết thúc

$\Delta$  : Tập hữu hạn các ký hiệu không kết thúc

$S \in \Delta$  : Ký hiệu đầu

$P$  : Tập hữu hạn các *sản xuất* , ký hiệu  $\alpha \rightarrow \beta$ ,  $\alpha$  chứa ít nhất 1 ký hiệu không kết thúc,  $\beta$  gồm ký hiệu kết thúc, ký hiệu không kết thúc và có thể không gồm ký hiệu nào.

## Ví dụ

$S \rightarrow \neg A | A$

$A \rightarrow B.B | B$

$B \rightarrow BC | C$

$C \rightarrow 0|1|2|3|4|5|6|7|8|9$

# Suy dẫn (Derivations)

- Mỗi lần thực hiện việc thay thế là một bước suy dẫn.
- Nếu mỗi dạng câu có nhiều ký hiệu không kết thúc để thay thế có thể sử dụng bất cứ sản xuất nào.

# Suy dẫn trái và suy dẫn phải

- Nếu giải thuật phân tích cú pháp chọn ký hiệu không kết thúc cực trái hay cực phải để thay thế, kết quả của nó là suy dẫn trái hoặc suy dẫn phải

# Cây suy dẫn(Cây phân tích cú pháp)

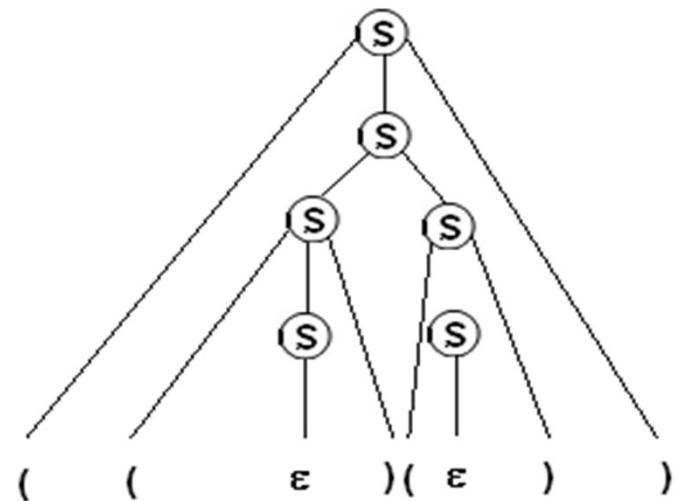
Cây suy dẫn có những đặc điểm sau

- 1) Mỗi nút của cây có nhãn là ký hiệu kết thúc, ký hiệu không kết thúc hoặc  $\varepsilon$  (xâu rỗng)
  - 2) Nhãn của nút gốc là S (ký hiệu đầu)
  - 3) Nút trong có nhãn là ký hiệu không kết thúc, nút lá có nhãn là ký hiệu kết thúc hoặc  $\varepsilon$
  - 4) Nút A có các nút con từ trái qua phải là  $X_1, X_2, \dots, X_k$  thì có một sản xuất dạng  $A -> X_1 X_2 \dots X_k$
  - 5) Nút lá có thể có nhãn  $\varepsilon$  chỉ khi tồn tại sản xuất  $A -> \varepsilon$  và nút cha của nút lá chỉ có một nút con duy nhất

Ví dụ : Cây suy dẫn của văn phạm

$S \rightarrow SS \mid (S) \mid \varepsilon$

Kết quả của cây PTCP có được bằng cách đọc nhãn các lá từ trái sang phải. Đó là một câu của văn phạm



# Văn phạm nhập nhằng

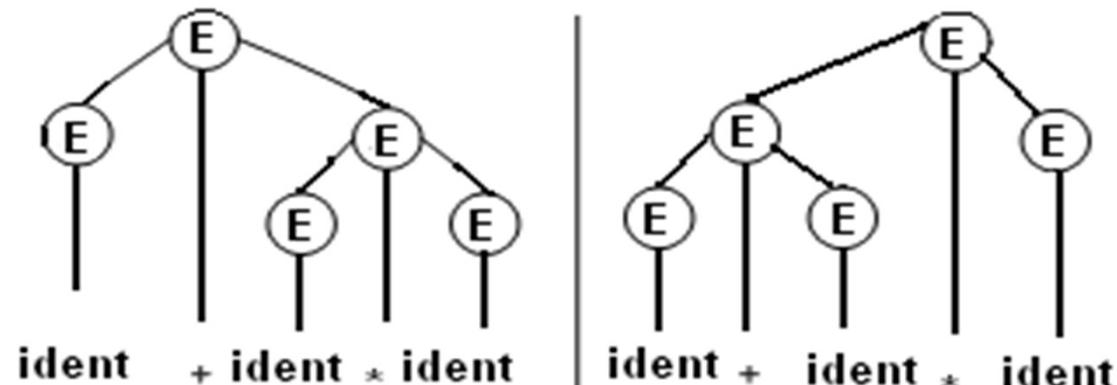
Văn phạm

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow ( E )$

$E \rightarrow \text{ident}$



Cho phép đưa ra hai suy diễn khác nhau cho  
xâu ident + ident \* ident (chẳng hạn  $x + y * z$ )

Văn phạm là nhập nhằng

# Khử nhập nhằng

$E \rightarrow E + T$

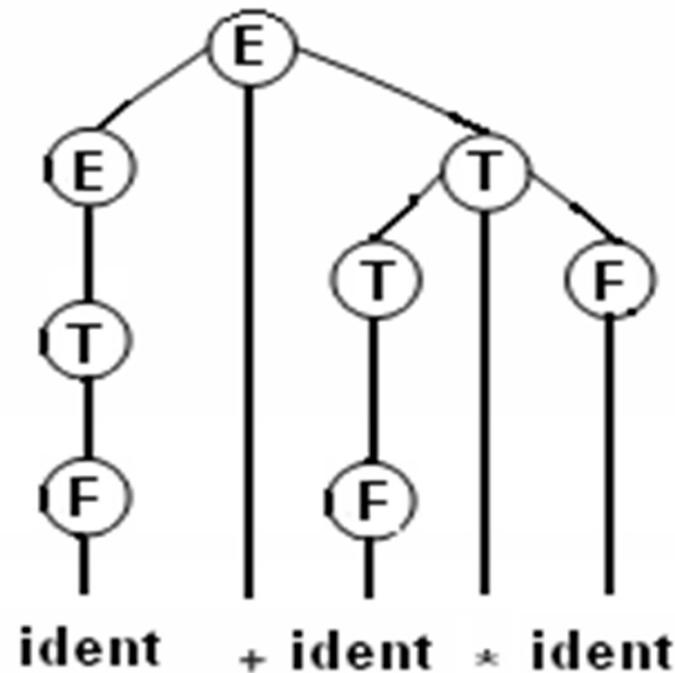
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow ( E )$

$F \rightarrow \text{ident}$



(Bằng cách thêm các ký hiệu không kết thúc và các sản xuất để đảm bảo thứ tự ưu tiên)

# Đệ quy

- **Một sản xuất là đệ quy nếu**  $X \Rightarrow^* \omega_1 X \omega_2$
- Có thể dùng để biểu diễn các quá trình lặp hay cấu trúc lồng nhau

Đệ quy trực tiếp  $X \Rightarrow^* \omega_1 X \omega_2$

Đệ quy trái  $X \Rightarrow b \mid X a. X \Rightarrow X a \Rightarrow X a a \Rightarrow X a a a \Rightarrow b a a a a a \dots$

Đệ quy phải  $X \Rightarrow b \mid a X. X \Rightarrow a a X \Rightarrow a a a X \Rightarrow \dots a a a a a b$

Đệ quy giữa  $X \Rightarrow b \mid "(" X ")" . X \Rightarrow (X) \Rightarrow ((X)) \Rightarrow (((X))) \Rightarrow (((\dots (b)\dots)))$

Đệ quy gián tiếp  $X \Rightarrow^* \omega_1 X \omega_2$

## Khử đệ quy trái

- Có thể loại các sản xuất đệ quy trái như sau: Thêm ký hiệu không kết thúc mới  $A'$ .  
Thay thế:

$$A \rightarrow A\alpha \mid \beta$$

bằng

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon.$$

# Khử đệ quy trái

$E \rightarrow E + T \mid T$

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

$F \rightarrow ( E ) \mid \text{ident}$

Khử đệ quy trái bằng cách thêm ký hiệu không  
kết thúc và sản xuất mới

$E \rightarrow T E'$

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

$T \rightarrow F T'$

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

$F \rightarrow ( E ) \mid \text{ident}$

## Công thức siêu ngữ Backus và các biến thể

- **Siêu ngữ (metalanguage)**: Ngôn ngữ sử dụng các lệnh để mô tả ngôn ngữ khác
- **BNF (Backus Naur Form)** là dạng siêu cú pháp để mô tả các ngôn ngữ lập trình
- BNF được sử dụng rộng rãi để mô tả văn phạm của các ngôn ngữ lập trình, tập lệnh và các giao thức truyền thông.

## Các biến thể của công thức siêu ngữ Backus

- Ký pháp BNF là một tập các luật, về trái của mỗi luật là một cấu trúc cú pháp.
- Tên của cấu trúc cú pháp được gọi là ký hiệu không kết thúc.
- Các ký hiệu không kết thúc thường được bao trong cặp <>.
- Các ký hiệu kết thúc thường được phân cách bằng cặp nháy đơn hoặc nháy kép

# Công thức siêu ngũ Backus và các biến thể

- Mỗi ký hiệu không kết thúc được định nghĩa bằng một hay nhiều luật.
- Các luật có dạng

$N ::= S$

*(N là ký hiệu không kết thúc, s là một xâu gồm 0 hay nhiều ký hiệu kết thúc và không kết thúc. Các luật có chung về trái được phân cách bằng | )*

## Ví dụ về BNF

```
<Số> ::= '-'<Số thập phân>| '+'<Số thập phân>|<số thập phân>  
<Số thập phân> ::= <Dãy chữ số>|<Dãy chữ số> '.'<Dãy chữ số>  
<Dãy chữ số> ::= <Chữ số>|<Chữ số><Dãy chữ số>  
<Chữ số> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

# EBNF

- EBNF (Extended BNF ) được phát triển từ ký pháp BNF. EBNF có ký pháp tương tự BNF nhưng được đơn giản hóa bằng cách sử dụng một số ký hiệu đặc biệt :
  - [] phần này là tuỳ chọn(có hoặc không)
  - { } phần này có thể lặp lại một số lần tuỳ ý hoặc không xuất hiện lần nào (Nếu lặp lại m hay n lần , dùng n hay m là chỉ số trên hoặc dưới)
  - Không cần dùng “ ” cho ký hiệu kết thúc

# So sánh BNF và EBNF

## Ví dụ

- Trong EBNF

*<Lệnh if> ::= IF <Biểu thức> THEN <Lệnh>  
[ELSE <Lệnh>]*

- Trong BNF

*<Lệnh if> ::= 'IF' <Biểu thức> 'THEN'  
<Lệnh> | 'IF' <Biểu thức> THEN <Lệnh>  
'ELSE' <Lệnh>*

# Văn phạm KPL biểu diễn bằng BNF

```
01) Prog ::= KW_PROGRAM Ident SB_SEMICOLON Block SB_PERIOD

02) Block ::= KW_CONST ConstDecl ConstDecls Block2
03) Block ::= Block2

04) Block2 ::= KW_TYPE TypeDecl TypeDecls Block3
05) Block2 ::= Block3

06) Block3 ::= KW_VAR VarDecl VarDecls Block4
07) Block3 ::= Block4

08) Block4 ::= SubDecls Block5
09) Block5 ::= KW_BEGIN Statements KW_END
```

# Văn phạm KPL biểu diễn bằng BNF

```
10) ConstDecls ::= ConstDecl ConstDecls
11) ConstDecls ::= ε

12) ConstDecl ::= Ident SB_EQUAL Constant SB_SEMICOLON

13) TypeDecls ::= TypeDecl TypeDecls
14) TypeDecls ::= ε

15) TypeDecl ::= Ident SB_EQUAL Type SB_SEMICOLON

16) VarDecls ::= VarDecl VarDecls
17) VarDecls ::= ε

18) VarDecl ::= Ident SB_COLON Type SB_SEMICOLON

19) SubDecls ::= FunDecl SubDecls
20) SubDecls ::= ProcDecl SubDecls
21) SubDecls ::= ε
```

# Văn phạm KPL biểu diễn bằng BNF

```
22) FunDecl   ::= KW_FUNCTION Ident Params SB_COLON BasicType SB_SEMICOLON
                  Block SB_SEMICOLON

23) ProcDecl   ::= KW PROCEDURE Ident Params SB_SEMICOLON Block SB_SEMICOLON

24) Params     ::= SB_LPAR Param Params2 SB_RPAR
25) Params     ::= ε

26) Params2    ::= SB_SEMICOLON Param Params2
27) Params2    ::= ε

28) Param      ::= Ident SB_COLON BasicType
29) Param      ::= KW_VAR Ident SB_COLON BasicType
```

# Văn phạm KPL biểu diễn bằng BNF

```
30) Type ::= KW_INTEGER
31) Type ::= KW_CHAR
32) Type ::= TypeIdent
33) Type ::= KW_ARRAY SB_LSEL Number SB_RSEL KW_OF Type

34) BasicType ::= KW_INTEGER
35) BasicType ::= KW_CHAR

36) UnsignedConstant ::= Number
37) UnsignedConstant ::= ConstIdent
38) UnsignedConstant ::= ConstChar

40) Constant ::= SB_PLUS Constant2
41) Constant ::= SB_MINUS Constant2
42) Constant ::= Constant2
43) Constant ::= ConstChar

44) Constant2 ::= ConstIdent
45) Constant2 ::= Number
```

# Văn phạm KPL biểu diễn bằng BNF

- 46) **Statements** ::= **Statement** **Statements2**
- 47) **Statements2** ::= **KW\_SEMICOLON** **Statement** **Statement2**
- 48) **Statements2** ::=  $\epsilon$
  
- 49) **Statement** ::= **AssignSt**
- 50) **Statement** ::= **CallSt**
- 51) **Statement** ::= **GroupSt**
- 52) **Statement** ::= **IfSt**
- 53) **Statement** ::= **WhileSt**
- 54) **Statement** ::= **ForSt**
- 55) **Statement** ::=  $\epsilon$

# Văn phạm KPL biểu diễn bằng BNF

```
56) AssignSt ::= Variable SB_ASSIGN Expression
57) AssignSt ::= FunctionIdent SB_ASSIGN Expression

58) CallSt    ::= KW_CALL ProcedureIdent Arguments

59) GroupSt   ::= KW_BEGIN Statements KW_END

60) IfSt       ::= KW_IF Condition KW_THEN Statement ElseSt

61) ElseSt     ::= KW_ELSE statement
62) ElseSt     ::= ε

63) WhileSt    ::= KW WHILE Condition KW DO Statement
64) ForSt      ::= KW FOR VariableIdent SB_ASSIGN Expression KW_TO
                  Expression KW DO Statement
```

# Văn phạm KPL biểu diễn bằng BNF

```
65) Arguments ::= SB_LPAR Expression Arguments2 SB_RPAR
66) Arguments ::= ε

67) Arguments2 ::= SB_COMMA Expression Arguments2
68) Arguments2 ::= ε

68) Condition ::= Expression Condition2

69) Condition2 ::= SB_EQ Expression
70) Condition2 ::= SB_NEQ Expression
71) Condition2 ::= SB_LE Expression
72) Condition2 ::= SB_LT Expression
73) Condition2 ::= SB_GE Expression
74) Condition2 ::= SB_GT Expression
```

# Văn phạm KPL biểu diễn bằng BNF

```
75) Expression ::= SB_PLUS Expression2
76) Expression ::= SB_MINUS Expression2
77) Expression ::= Expression2

78) Expression2 ::= Term Expression3

79) Expression3 ::= SB_PLUS Term Expression3
80) Expression3 ::= SB_MINUS Term Expression3
81) Expression3 ::= ε

82) Term ::= Factor Term2

83) Term2 ::= SB_TIMES Factor Term2
84) Term2 ::= SB_SLASH Factor Term2
85) Term2 ::= ε
```

# Văn phạm KPL biểu diễn bằng BNF

```
86) Factor ::= UnsignedConstant
87) Factor ::= Variable
88) Factor ::= FunctionApptication
89) Factor ::= SB_LPAR Expression SB_RPAR

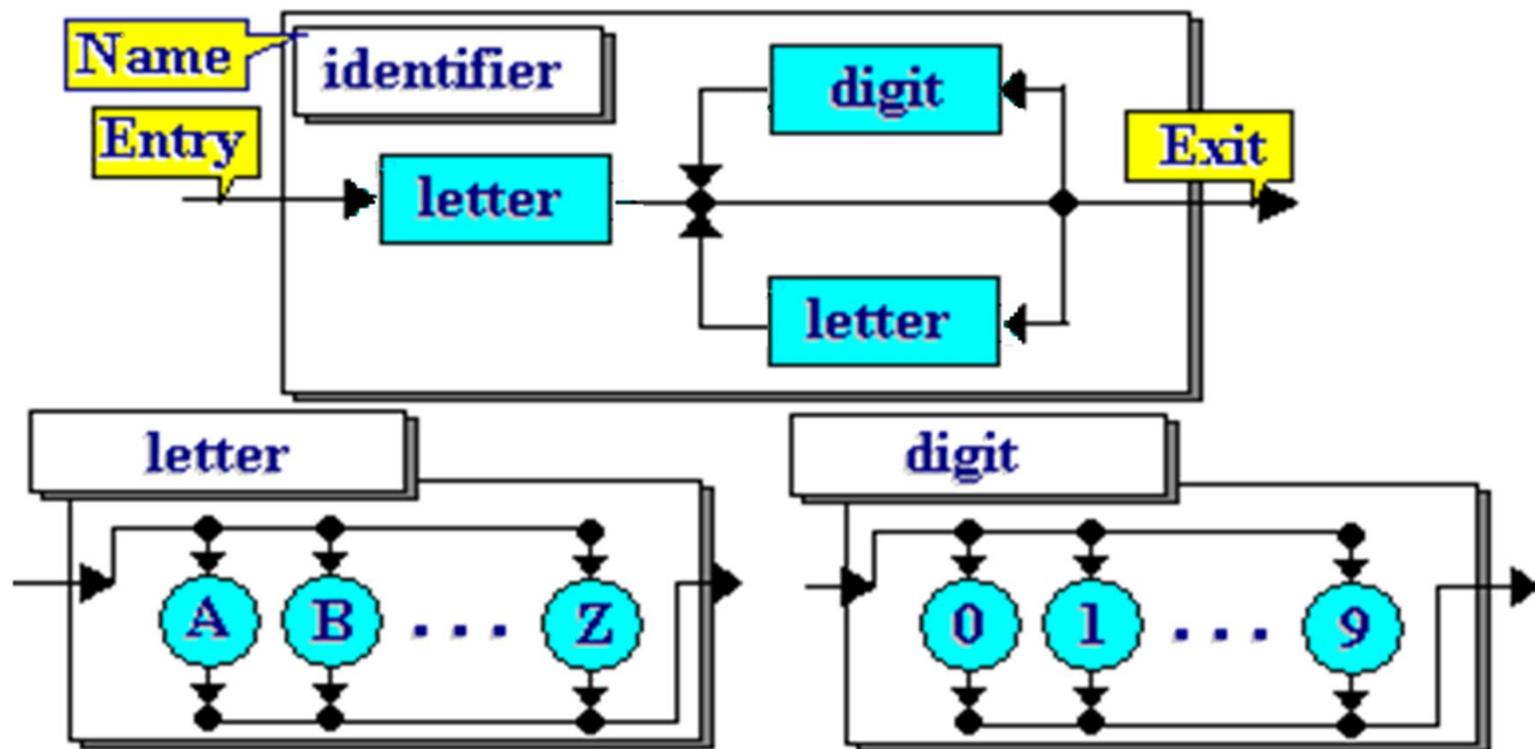
90) Variable ::= VariableIdent Indexes
91) FunctionApplication ::= FunctionIdent Arguments

92) Indexes ::= SB_LSEL Expression SB_RSEL Indexes
93) Indexes ::= ε
```

# Sơ đồ cú pháp

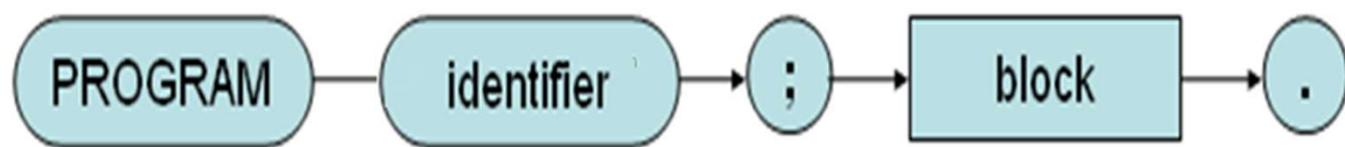
- Là công cụ để mô tả cú pháp của ngôn ngữ lập trình dưới dạng đồ thị
- Mỗi sơ đồ cú pháp là một đồ thị định hướng với lối vào và lối ra xác định.
- Mỗi sơ đồ cú pháp có một tên duy nhất
- Hình tròn (oval): ký hiệu kết thúc
- Hình chu nhات: ký hiệu không kết thúc

# Ví dụ một sơ đồ cú pháp

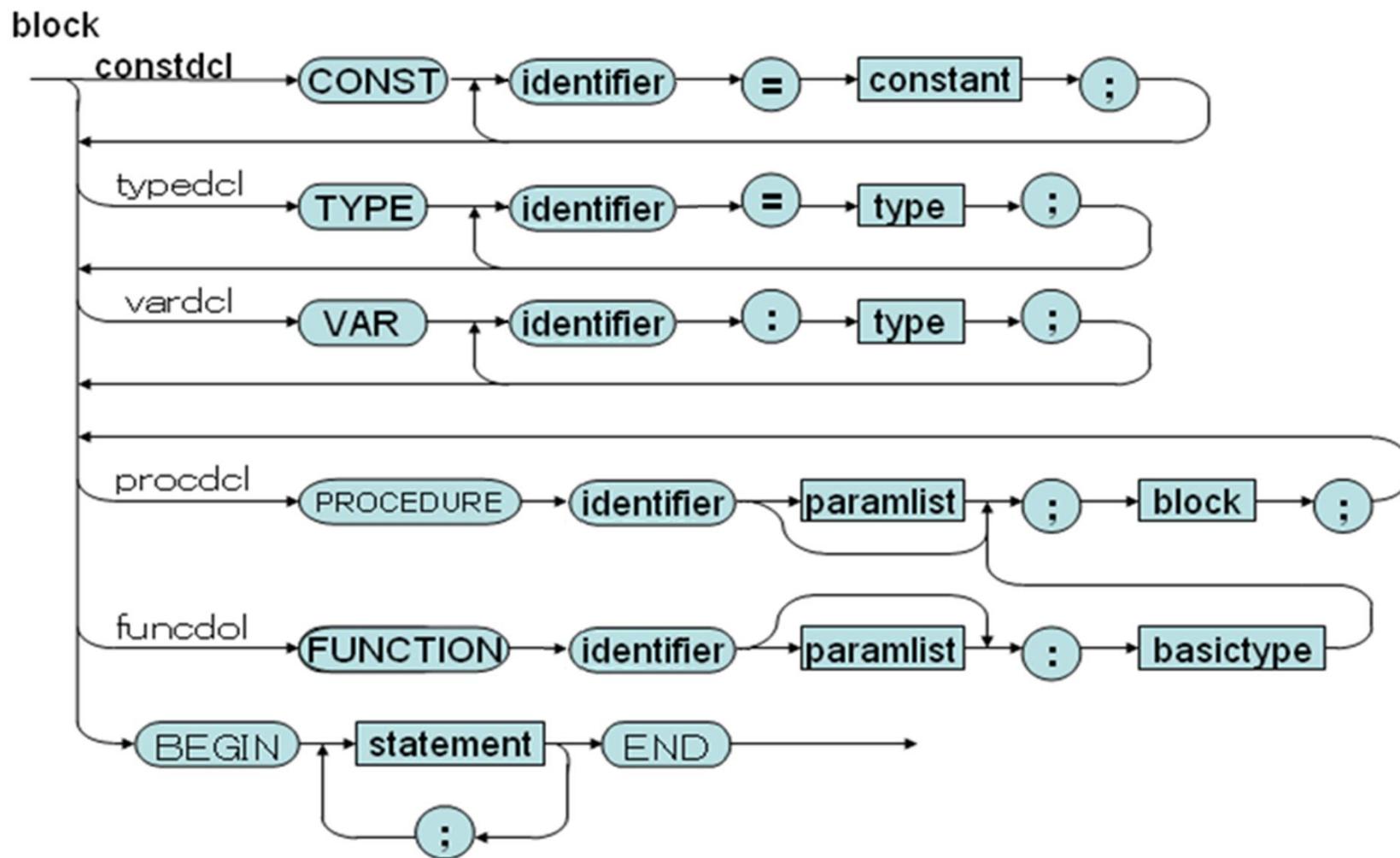


# Sơ đồ cú pháp của KPL (Tổng thể CT)

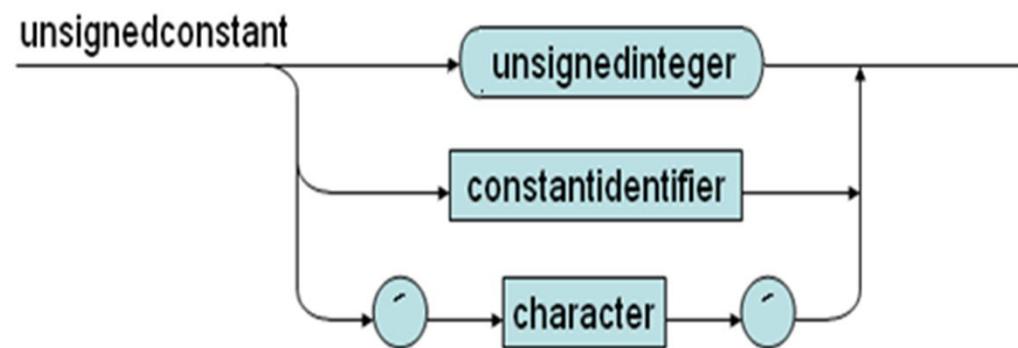
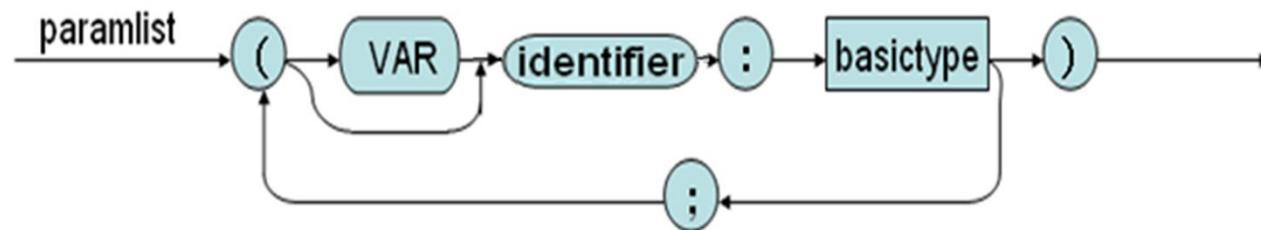
program



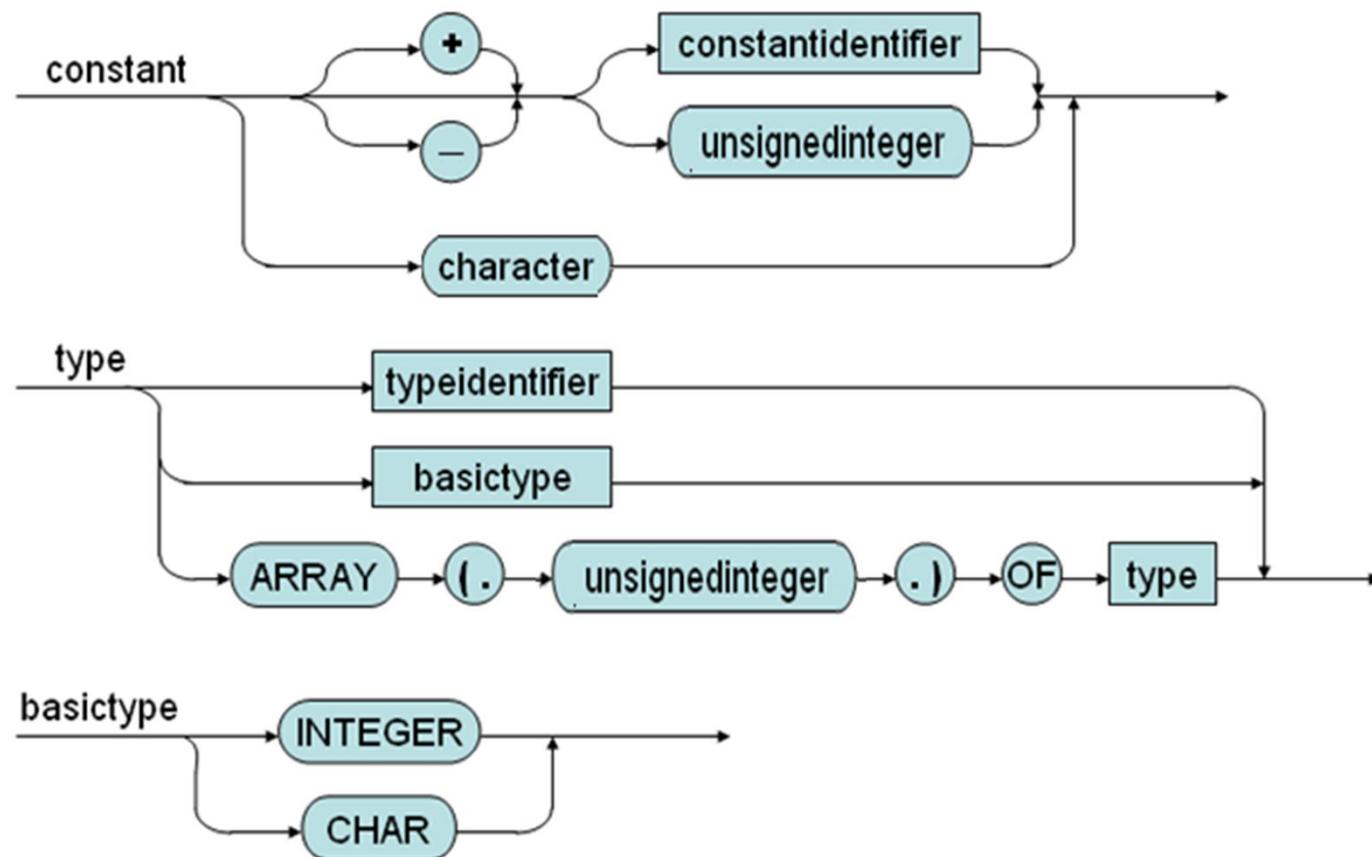
# Sơ đồ cú pháp của KPL (Khối)



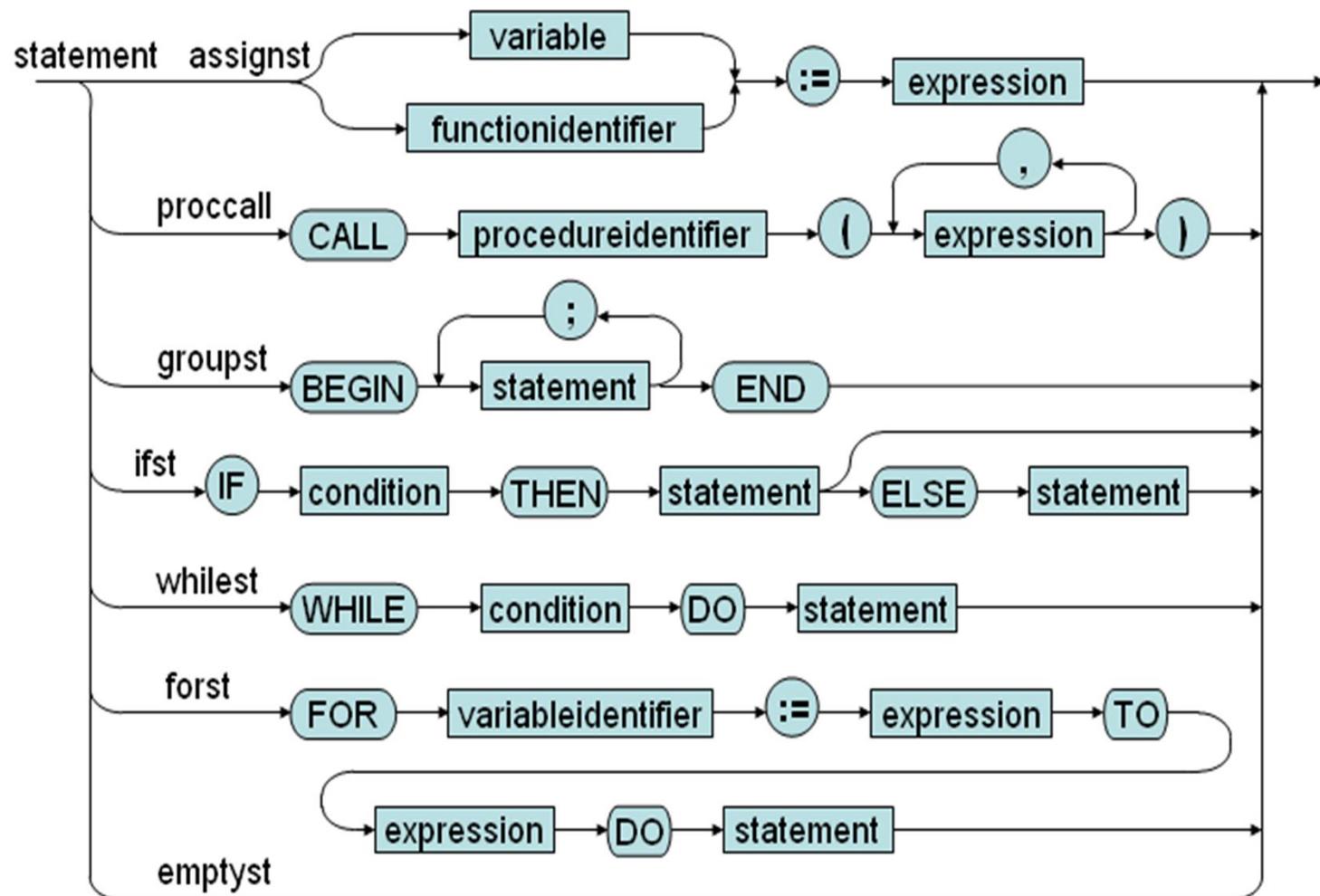
# Sơ đồ cú pháp của KPL(tham số, hằng không dấu)



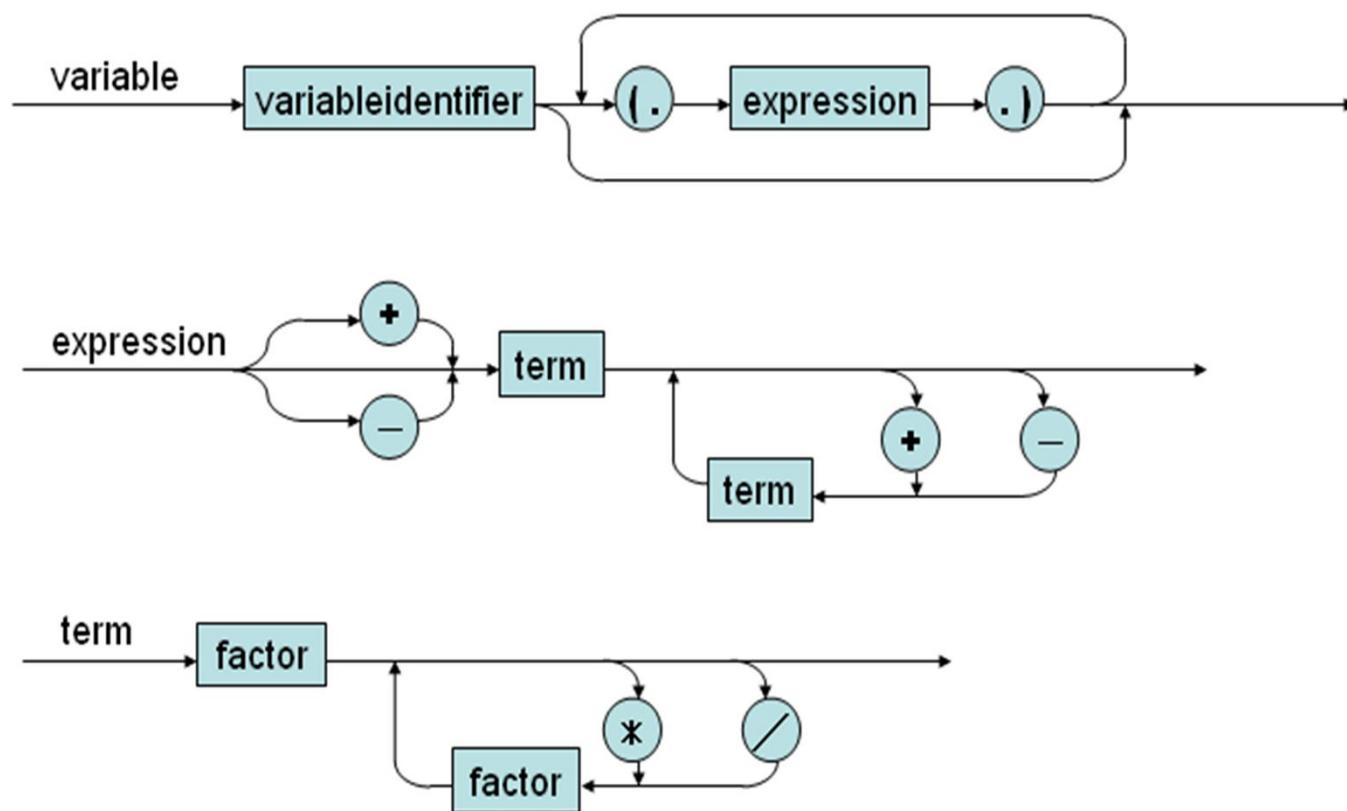
# Sơ đồ cú pháp của KPL (Khai báo)



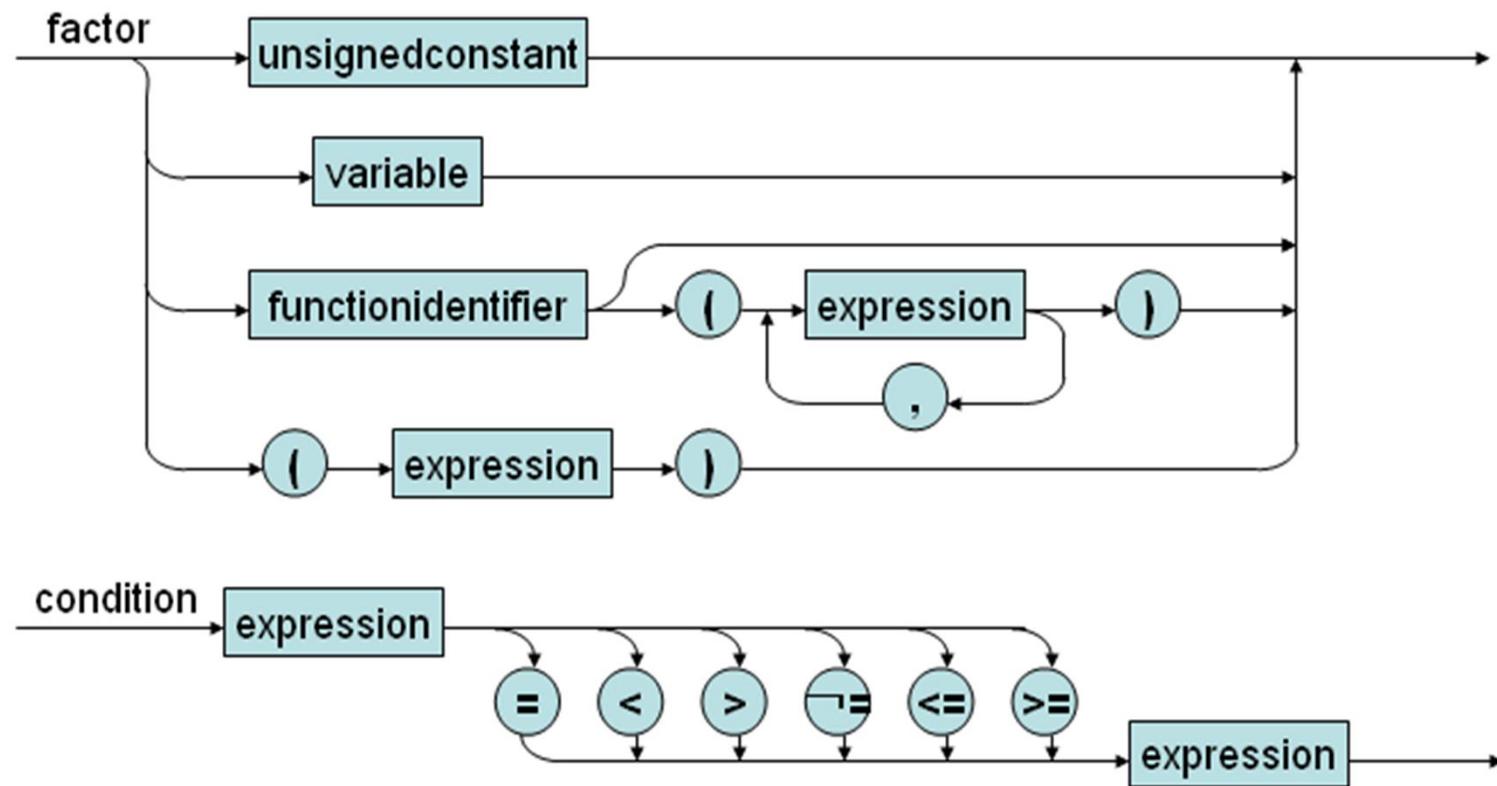
# Sơ đồ cú pháp của KPL (lệnh)



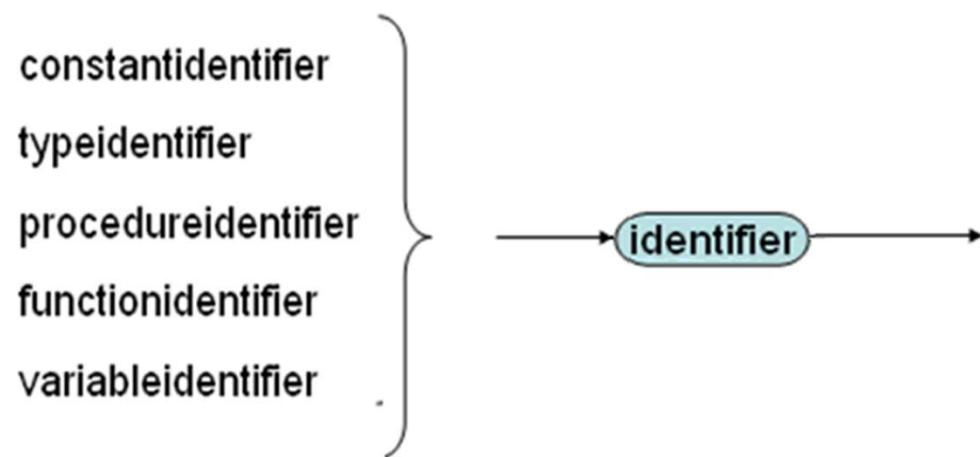
# Sơ đồ cú pháp của KPL (biểu thức)



# Sơ đồ cú pháp của KPL (nhân tử, điều kiện)



# Sơ đồ cú pháp của KPL (các loại tên)



# Bài toán phân tích cú pháp

Bài toán đặt ra

*Cho văn phạm phi ngũ cảnh  $G$  và xâu  $w$   
 $w \in L(G)$  đúng hay sai?*

Phân tích trên xuống (top down)

$S \Rightarrow^* w?$

# w đúng cú pháp $\Rightarrow$ cây phân tích cú pháp

$E \rightarrow E + T$

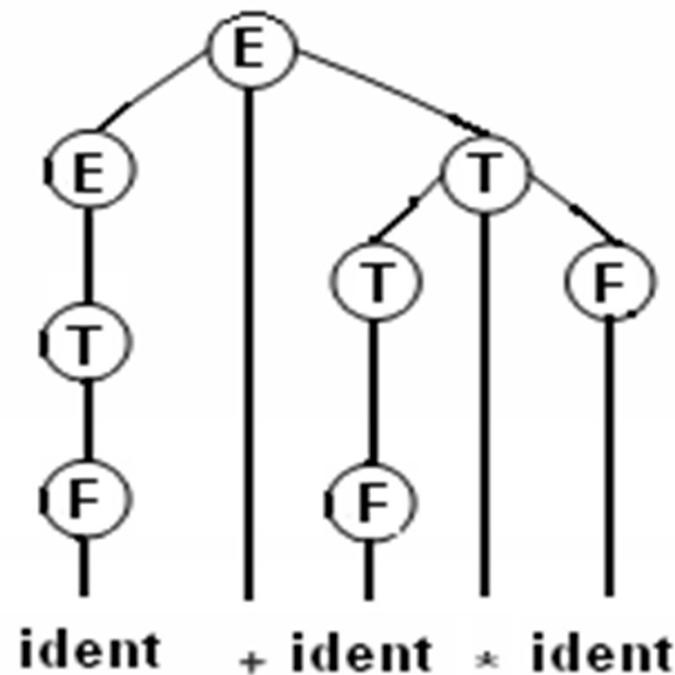
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow ( E )$

$F \rightarrow \text{ident}$



Biểu diễn cây phân tích cú pháp bằng cách nào?

# Phân tích trái, phân tích phải

- *Phân tích trái* của  $\alpha$  là dãy các sản xuất được sử dụng trong suy dẫn trái ra  $\alpha$  từ  $S$
- *Phân tích phải* của  $\alpha$  là nghịch đảo của dãy các sản xuất được sử dụng trong suy dẫn phải ra  $\alpha$  từ  $S$
- *Phân tích là danh sách các số* từ 1 đến  $p$

## Ví dụ

- Xét văn phạm G, các sản xuất được đánh số như sau
  - 1.  $E \rightarrow T+E$
  - 2.  $E \rightarrow T$
  - 3.  $T \rightarrow F^* T$
  - 4.  $T \rightarrow F$
  - 5.  $F \rightarrow (E)$
  - 6.  $F \rightarrow a$
- Phân tích trái của xâu  $a^*(a+a)$  là 23645146246

## Chương 2: Các phương pháp chung để PTCP

### 2.1. Các phương pháp phân tích quay lui

- Phân tích trên xuống
- Phân tích dưới lên

### 2.2. Các phương pháp phân tích bảng

- Giải thuật CYK
- Giải thuật Earley

# Bài toán đặt ra

*Cho văn phạm phi ngữ cảnh  $G$  và xâu  $w$   
 $w \in L(G)$  đúng hay sai?*

Phân tích trên xuống (top down)

$S \Rightarrow^* w?$

Phân tích dưới lên (bottom up)

$S \Leftarrow^* w?$

Phân tích bảng

# w đúng cú pháp $\Rightarrow$ cây cú pháp

$E \rightarrow E + T$

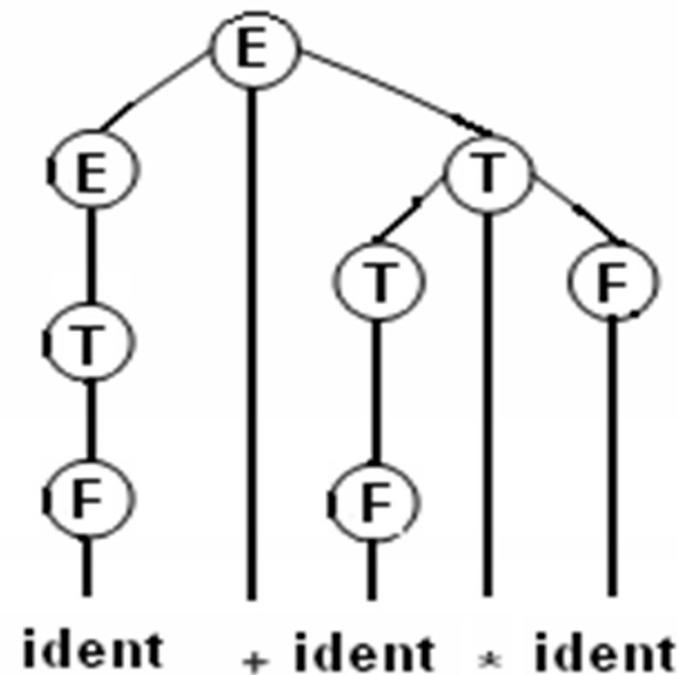
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow ( E )$

$F \rightarrow \text{ident}$



Biểu diễn cây cú pháp bằng cách nào?

# Phân tích trái và phân tích phải

- *Phân tích trái* của  $\alpha$  là dãy các sản xuất được sử dụng trong suy dẫn trái ra  $\alpha$  từ  $S$
- *Phân tích phải* của  $\alpha$  là nghịch đảo của dãy các sản xuất được sử dụng trong suy dẫn phải ra  $\alpha$  từ  $S$
- *Phân tích là danh sách* các số từ 1 đến  $p$

## Ví dụ

Xét văn phạm G, các sản xuất được đánh số như sau

1.  $E \rightarrow T+E$
2.  $E \rightarrow T$
3.  $T \rightarrow F^* T$
4.  $T \rightarrow F$
5.  $F \rightarrow (E)$
6.  $F \rightarrow a$

Suy dẫn trái:  $E \Rightarrow T \Rightarrow F^* T \Rightarrow a^* T \Rightarrow a^* F \Rightarrow a^*(E) \Rightarrow a^*(T+E) \Rightarrow a^*(F+E) \Rightarrow a^*(a+E) \Rightarrow a^*(a+T) \Rightarrow a^*(a+F) \Rightarrow a^*(a+a)$

- Phân tích trái của xâu  $a^*(a+a)$  là 23645146246
- Phân tích phải của xâu  $a^*(a+a)$  là 66464215432

## 2.1. Các phương pháp phân tích quay lui

- Phân tích trên xuống (top down)
- Xây dựng cây phân tích cú pháp bắt đầu từ nút gốc S. Các nút con được tạo ra một cách đệ quy.
- Phân tích dưới lên (bottom up)  
Xuất phát từ các nút lá , thử xây dựng cây cú pháp cho đến tận nút gốc.

# Giải thuật phân tích top down quay lui

- Tư tưởng chủ yếu của giải thuật là xây dựng cây phân tích cú pháp (cây suy diễn) cho xâu w
- Đánh số thứ tự các sản xuất có cùng về phải, như vậy, các A - sản xuất của văn phạm sẽ được xếp thứ tự

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

## Mô tả giải thuật

- Bắt đầu từ nút gốc S
- Nút S được coi là nút hoạt động
- Tạo ra các nút con một cách đệ quy

## Nút hoạt động có nhãn là ký hiệu không kết thúc A

- Chọn vế phải đầu tiên của A- sản xuất :  $X_1 X_2 \dots X_k$ .
- Tạo k nút con trực tiếp của A với nhãn  $X_1, X_2, \dots, X_k$ .
- Nút hoạt động là nút nhãn  $X_1$ .
- Nếu  $k = 0$ , (sản xuất  $A \rightarrow \varepsilon$ ) thì nút hoạt động sẽ là nút bên phải (ngay sau) A khi duyệt cây theo thứ tự trái

# Nút hoạt động có nhãn là ký hiệu kết thúc a

- So sánh với ký hiệu đang xét.
  - Nếu trùng với ký hiệu đang xét thì chuyển đầu đọc sang phải 1 ô, chuyển sang xét nút bên phải.
  - Nếu a không trùng với ký hiệu đang xét thì quay lui tới nút mà tại đó đã sử dụng sản xuất trước (Thay thế một ký hiệu không kết thúc (chẳng hạn A) bằng về phải một sản xuất).
  - Chuyển đầu đọc sang trái (nếu cần) và thử với lựa chọn tiếp theo. Nếu không còn lựa chọn nào khác thì quay lui tới bước trước đó
- Nếu đã quay lui tới S và không còn lựa chọn khác: câu sai cú pháp

## Điều kiện để thực hiện giải thuật

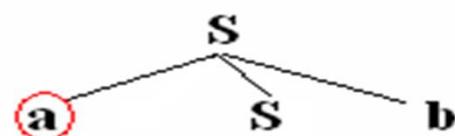
*Văn phạm G cần thoả điều kiện  
không để quy trái để tránh rơi vào  
chu trình*

## Ví dụ

- Cho văn phạm  
$$S \rightarrow aSb \mid c$$
- Xét xâu vào aacbb

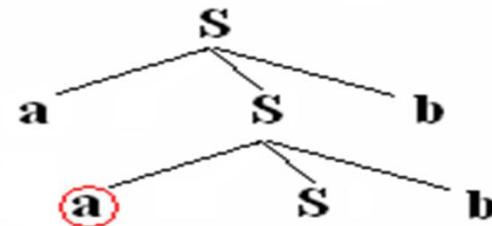
# Dụng cây phân tích cú pháp

a	a	c	b	b	EOF
---	---	---	---	---	-----



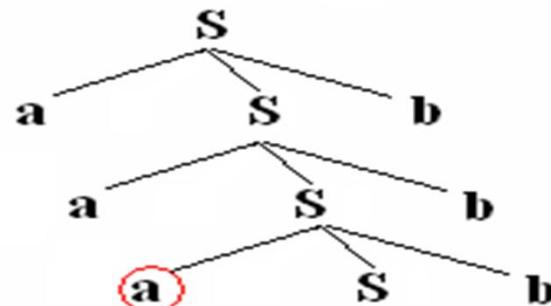
(a)

a	a	c	b	b	EOF
---	---	---	---	---	-----



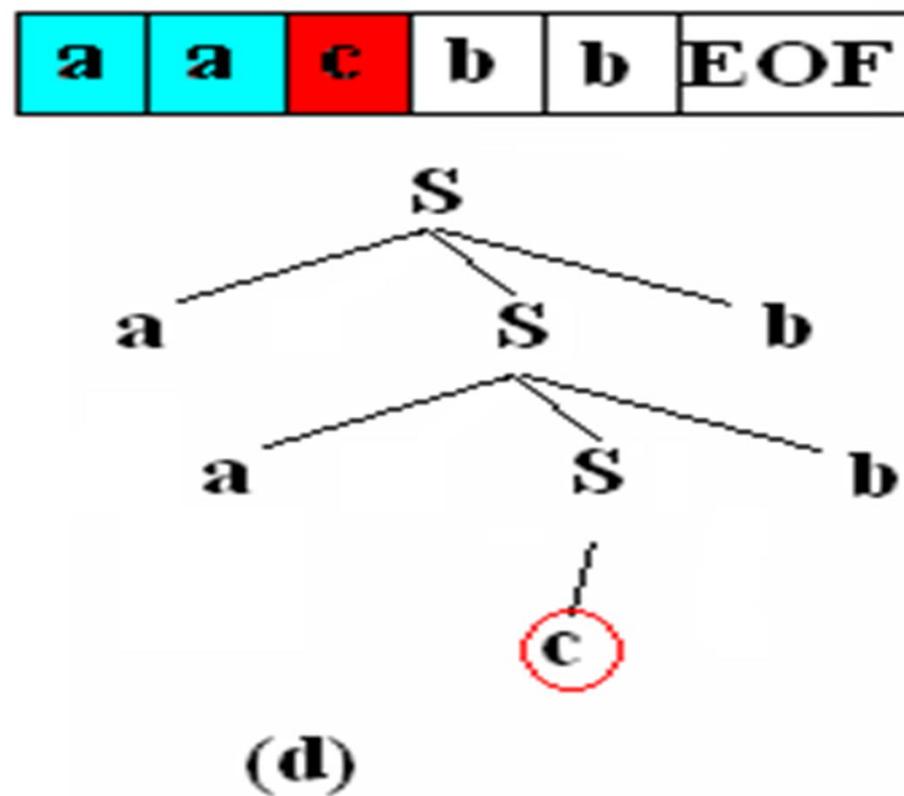
(b)

a	a	c	b	b	EOF
---	---	---	---	---	-----



(c)

# Thủ lựa chọn khác



# Giải thuật phân tích cú pháp trên xuống quay lui

- Vào

Văn phạm G phi ngũ cảnh không đệ quy trái,

xâu  $w = a_1 \dots a_n$ ,  $n \geq 0$

Các sản xuất của G được đánh số 1, . . . , q

- Ra

Một phân tích trái cho  $w$  (nếu có)

Thông báo lỗi nếu ngược lại

# Phương pháp

*Xây dựng 2 stack  $D_1$  và  $D_2$ .*

*$D_2$  biểu diễn dạng câu trái hiện tại có được bằng cách thay thế các ký hiệu không kết thúc bởi vé phải tương ứng*

*$D_1$  ghi lại lịch sử những lựa chọn đã sử dụng và những ký hiệu vào trên đó đầu đọc đã đổi vị trí*

## Đánh số các sản xuất chung về trái

- $\forall A \in N$  , giả sử có các A-sản xuất

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

Coi các sản xuất trên là

$$A_1 \rightarrow \alpha_1$$

.....

$$A_n \rightarrow \alpha_n$$

# Hình trạng của giải thuật

## Bộ bốn ( $s$ , $i$ , $\alpha$ , $\beta$ )

- $s \in Q$ : Trạng thái hiện thời
  - »  $q$ : Trạng thái bình thường
  - »  $b$ : Quay lui
  - »  $t$ : Kết thúc
- $i$ : Vị trí đầu đọc (Băng vào có dấu hiệu kết thúc #)
- $\alpha$ : Nội dung stack thứ nhất
- $\beta$ : Nội dung stack thứ hai

# Thực hiện giải thuật

- Bắt đầu từ hình trạng đầu, tính liên tiếp các hình trạng tiếp theo cho đến khi không tính được nữa.
- Nếu hình trạng cuối là  $(t, n+1, \gamma, \varepsilon)$ , đưa ra  $h(\gamma)$  và dừng. Ngược lại đưa ra thông báo sai

## Ví dụ

- Xét xâu vào aacbb và văn phạm G với các sản xuất

$S \rightarrow aSb$

$S \rightarrow c$

# Đánh số lại các sản xuất

1.  $S_1 \rightarrow aSb$
2.  $S_2 \rightarrow c$

# Quá trình thay đổi hình trạng

- $(q, 1, \varepsilon, S\#)$
- $\vdash (q, 1, S_1, aSb\#)$
- $\vdash (q, 2, S_1a, Sb\#)$
- $\vdash (q, 2, S_1aS_1, aSbb\#)$
- $\vdash (q, 3, S_1aS_1a, Sbb\#)$
- $\vdash (q, 3, S_1aS_1aS_1, aSbbb\#)$
- $\vdash (b, 3, S_1aS_1aS_1, aSbbb\#)$
- $\vdash (q, 3, S_1aS_1aS_2, cbb\#)$
- $\vdash (q, 4, S_1aS_1aS_2c, bb\#)$
- $\vdash (q, 5, S_1aS_1aS_2cb, b\#)$
- $\vdash (q, 6, S_1aS_1aS_2cbb, \#)$
- $\vdash (t, 6, S_1aS_1aS_2cbb, \varepsilon )$

## Tìm phân tích trái

$h(a) = \varepsilon$   $\forall a$  là ký hiệu kết thúc

$h(A_i) = p$ ,  $p$  là số hiệu của sản xuất liên hệ với sản xuất  $A \rightarrow \gamma$  với  $\gamma$  là lựa chọn thứ  $i$  của  $A$

*Ví dụ: Trong văn phạm*

1.  $S_1 \rightarrow aSb$
2.  $S_2 \rightarrow c$

$h(S_1aS_1aS_2cbb) = 112$

# Thử phân tích trên xuống quay lui với KPL

- Phân tích từ vựng và mã hóa từ tố
- Tập sản xuất của văn phạm

# Chuyển sơ đồ cú pháp thành luật

`<program > ::= program ident ; <block>.`

`<block> ::= <const-decl><type-decl>  
<proc-decl><func-decl><var-decl> begin  
  <statement-list> end`

# Mã hóa ký hiệu không kết thúc

```
if(str=="<program>") return 1;  
if(str=="<block>") return 2;  
if(str=="<const-decl>") return 3;  
if (str == "<const-assign-list>") return  
    4;  
if (str == "<constant>") return 5;  
if(str=="<type-decl>") return 6;  
if (str == "<type-assign-list>") return 7;  
if (str == "<type>") return 8;  
if (str == "<basictype>") return 9;  
if(str=="<var-decl>") return 10;  
if (str == "<ident-list>") return 11;  
if(str=="<proc-decl>") return 12;
```

```
if (str == "<para-list>") return 13;  
if (str == "<para-one>") return 14;  
if(str=="<func-decl>") return 15;  
if(str=="<statement-list>") return 16;  
if(str=="<statement>") return 17;  
if (str == "<condition>") return 18;  
if (str == "<relation>") return 19;  
if(str=="<expression>") return 20;  
if (str == "<adding-op>") return 21;  
if(str=="<term>") return 22  
if(str=="<multiplying-op>") return 23;  
if (str == "<factor>") return 24;
```

# Mã hóa từ tố: tên, số, hằng ký tự

```
// ident;  
if(str == "ident") return 25;  
//const  
if(str == "number")return 26;  
if (str == "charcon") return 27;  
//operator  
if(str == "plus")return 28;  
if (str == "minus") return 29;  
if (str == "times") return 30;  
if (str == "slash") return 31;  
if (str == "oddsym") return 32;  
if (str == "assign") return 33;  
if (str == "leq") return 34;
```

```
//specific symbol  
if (str == "lparen") return 35;  
if (str == "rparen") return 36;  
if (str == "comma") return 37;  
if (str == "semicolon") return 38;  
if (str == "period") return 39;  
if (str == "becomes") return 40;  
if (str == "lbrace") return 41;  
if (str == "rbrace") return 42;  
if (str == "lbrack") return 43;  
if (str == "rbrack") return 44;
```

# Mã hóa từ tố: từ khóa

```
if (str == "beginsym") return 45; if (str == "varsym") return 53;  
if (str == "endsym") return 46; if (str == "progsym") return 54;  
if (str == "ifsym") return 47; if (str == "funcsym") return 55;  
if (str == "thensym") return 48; if (str == "typesym") return 56;  
if (str == "whilesym") return 49; if (str == "arraysym") return 57;  
if (str == "dosym") return 50; if (str == "ofsym") return 58;  
if (str == "callsym") return 51; if (str == "intsym") return 59;  
if (str == "constsym") return 52; if (str == "charsym") return 60;
```

# Mã hóa từ tố: phép toán quan hệ

```
//relations  
if (str == "eql") return 61;  
if (str == "leq") return 62;  
if (str == "neq") return 63;  
if (str == "lss") return 64;  
if (str == "gtr") return 65;  
if (str == "geq") return 66;
```

# Mã hóa sản xuất

`<program > ::= program ident ; <block>.`

`setlaw[1,1]="54 25 38 2 39 ";`

`<block> ::= <const-decl><type-decl>  
<proc-decl><func-decl><var-decl> begin <statement-list> end`

`setlaw[2,1]=" 3 6 12 15 10 45 16 46 ";`

## Nhân xét

- Cài đặt phức tạp
- Chi phí thời gian quá lớn nếu chương trình phải phân tích gồm nhiều ký hiệu (tùy tố)
- Không thể thông báo lỗi chi tiết

## Phân tích cú pháp từ dưới lên (bottom up)

- Cách thức: gạt - thu gọn.
- Duyệt qua tất cả các suy dẫn phải để tìm ra một suy dẫn phù hợp với xâu đưa vào.

# Một dịch chuyển bao gồm:

- Tìm về phải sản xuất phù hợp với xâu ở đỉnh stack.
  - Nếu tìm thấy, thay thế các ký hiệu xuất hiện ở về phải bằng về trái tương ứng.
  - Nếu có nhiều sản xuất thoả mãn thì chọn lựa chọn thứ nhất.
- Nếu không còn thu gọn nào có thể áp dụng, gạt ký hiệu vào tiếp theo vào danh sách đầy xuống và xử lý như trên.
- Nếu đến cuối xâu mà không thu gọn nào có thể sử dụng, quay lại dịch chuyển cuối cùng đã thực hiện thu gọn. Thủ khả năng thu gọn khác (nếu có)

# Ví dụ minh họa

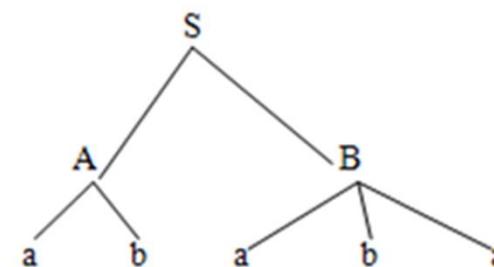
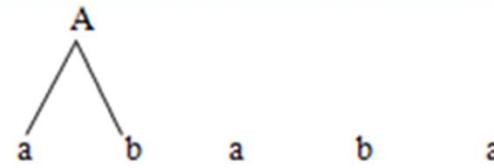
Văn phạm

$S \rightarrow AB$

$A \rightarrow ab$

$B \rightarrow aba$

Xâu vào ababa



## Hình trạng của giải thuật

(s, i,  $\alpha$ ,  $\beta$ )

s: trạng thái hiện thời (q, b, t).

i: vị trí đầu đọc trên xâu vào.

$\alpha$ : nội dung danh sách  $D_1$  (chứa xâu các ký hiệu suy dẫn ra phần xâu vào tính đến bên trái của đầu đọc

$\beta$ : nội dung danh sách  $D_2$  lưu trữ lịch sử gạt – thu gọn cần thiết để đạt được nội dung của danh sách  $D_1$

# Hình trạng của giải thuật

- Hình trạng đầu:  $(q, 1, \#, \varepsilon)$
- Hình trạng cuối khi xâu được đoán nhận  
 $(t, n+1, \#S, \gamma)$   
Vị trí thứ  $n+1$  để lưu trữ giới hạn phải  $\#$
- Đánh giá  $h(\gamma)$ :
  - $h(s) = \varepsilon$
  - $h(j) = j$  với mọi sản xuất
  - $h(\gamma)$  là nghịch đảo của phân tích phải của  $w$

## Ví dụ

### Văn phạm G với các sản xuất

1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T^* F$
4.  $T \rightarrow F$
5.  $F \rightarrow a$

Xâu vào  $a^*a$

# Quá trình thay đổi hình trạng với xâu $a^*a$

$(q, 1, \#, \varepsilon)$	$\vdash (b, 4, \#E^*a, ss245s)$
$\vdash (q, 2, \#a, s)$	$\vdash (b, 3, \#E^*, s245s)$
$\vdash (q, 2, \#F, 5s)$	$\vdash (b, 2, \#E, 245s)$
$\vdash (q, 2, \#T, 45s)$	$\vdash (b, 2, \#T, 45s)$
$\vdash (q, 2, \#E, 245s)$	$\vdash (q, 3, \#T^*, s45s)$
$\vdash (q, 3, \#E^*, s245s)$	$\vdash (q, 4, \#T^*a, ss45s)$
$\vdash (q, 4, \#E^*a, ss245s)$	$\vdash (q, 4, \#T^*F, 5ss45s)$
$\vdash (q, 4, \#E^*F, 5ss245s)$	$\vdash (q, 4, \#T, 35ss45s)$
$\vdash (q, 4, \#E^*T, 45ss245s)$	$\vdash (q, 4, \#E, 235 ss 45s)$
$\vdash (q, 4, \#E^*E, 245ss245s)$	$\vdash (t, 4, \#E, 235 ss 45s)$
$\vdash (b, 4, \#E^*E, 245ss245s)$	$h(235ss45s)=23545$
$\vdash (b, 4, \#E^*T, 45ss245s)$	Phân tích phải tìm được 54532
$\vdash (b, 4, \#E^*F, 5ss245s)$	

## 2.2. Các phương pháp phân tích bảng

- Phân tích được toàn bộ các ngôn ngữ phi ngữ cảnh
- Độ phức tạp  $O(n^3)$
- Thường dùng để phân tích câu trong ngôn ngữ tự nhiên
- Giải thuật CYK
- Giải thuật Earley

# Phân tích bảng: quy hoạch động

- Tạo phân tích bảng cho các cây con của cây phân tích cú pháp trong quá trình xây dựng phân tích tổng thể của câu
- Tìm kiếm trên các cây con thay vì phân tích lại như trong giải thuật quay lui.
- Có khả năng giải quyết vấn đề nhập nhằng

# Giải thuật CYK (Cocke-Younger-Kasami)

- Sử dụng để phân tích cú pháp cho các văn phạm ở dạng chuẩn Chomsky

## Dạng chuẩn Chomsky

Mọi sản xuất có dạng

- $X \rightarrow YZ$
- $X \rightarrow a$
- $S \rightarrow \epsilon$  ( $S$  không xuất hiện ở về phải SX khác)

## Ví dụ

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow CC \mid a \mid c \\ B &\rightarrow BC \mid b \\ C &\rightarrow CB \mid BA \mid c \end{aligned}$$

# Giải thuật CYK

- **Tư tưởng chính**

Cho xâu s độ dài N

For  $k = 1$  to  $N$

Với mọi xâu con độ dài  $k$

Tìm mọi ký hiệu không kết thúc suy dẫn ra nó.

# Giải thuật CYK – Ví dụ

- **Văn phạm phi ngũ cành**

(1)	$S \rightarrow AB$
(2, 3, 4)	$A \rightarrow CC \mid a \mid c$
(5, 6)	$B \rightarrow BC \mid b$
(7, 8, 9)	$C \rightarrow CB \mid BA \mid c$

- Xâu cần phân tích **abbc**

# Minh họa giải thuật CYK (1)

- Tư tưởng: Phân tích xâu theo thứ tự sau:

- Độ dài 1 với các xâu con

abbc

abbc

abbc

abbc

- Độ dài 2 với các xâu con

abbc

abbc

abbc

## Minh họa giải thuật CYK (2)

- Độ dài 3 với các xâu con

abbc

abbc

- Độ dài 4 với xâu

abbc

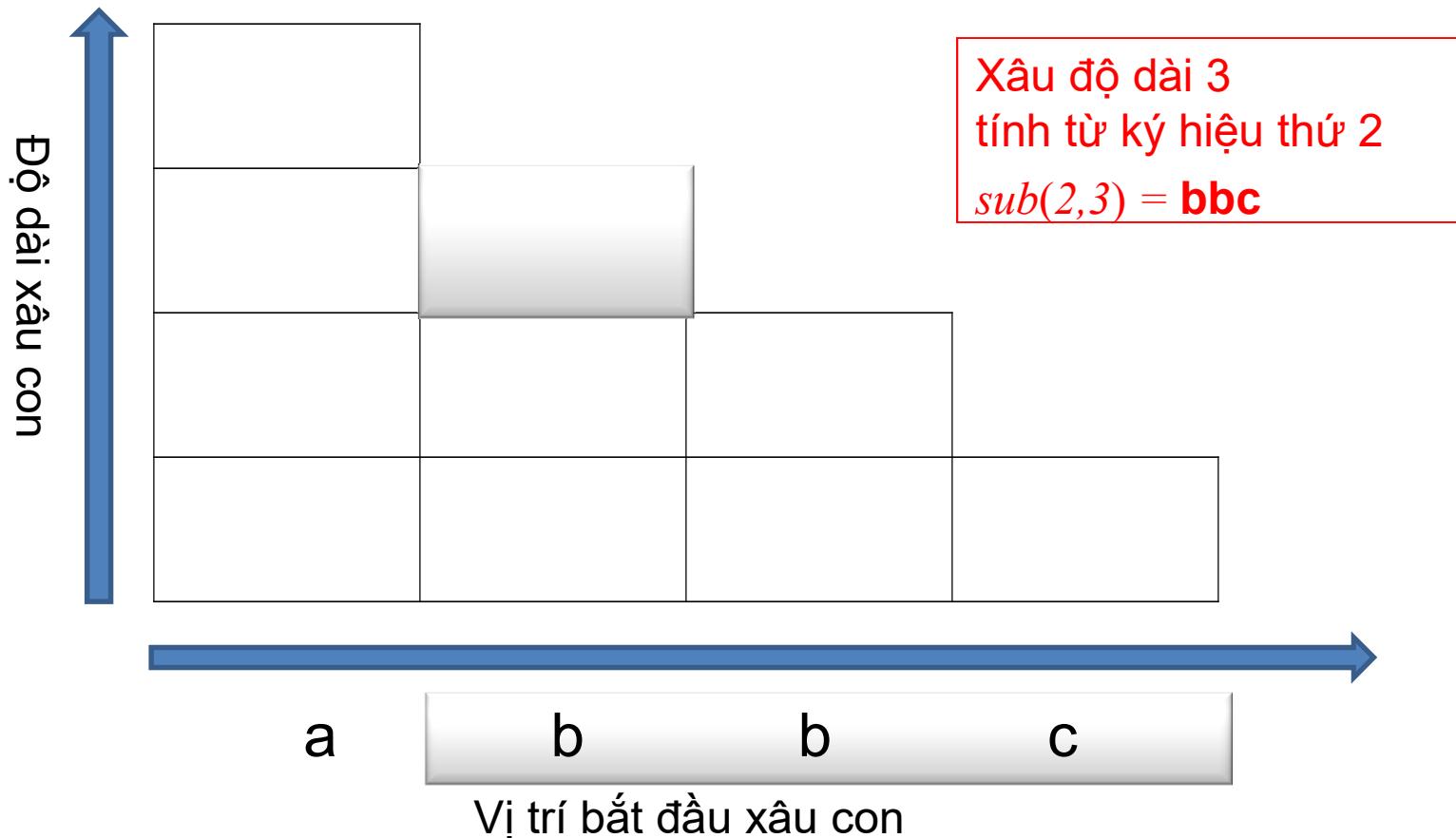
- Hoàn thành!

# Tư tưởng chính của giải thuật CYK

- Tư tưởng chính: Phân tích cú pháp xâu dài dựa trên phân tích cú pháp của xâu ngắn hơn.
- Ví dụ: **abb** phân tách thành  
**ab** và **b**  
**a** và **bb**
- Nếu đã biết phân tích của **ab** và **b** (hoặc, **a** và **bb**) ta có thể phân tích **abb**

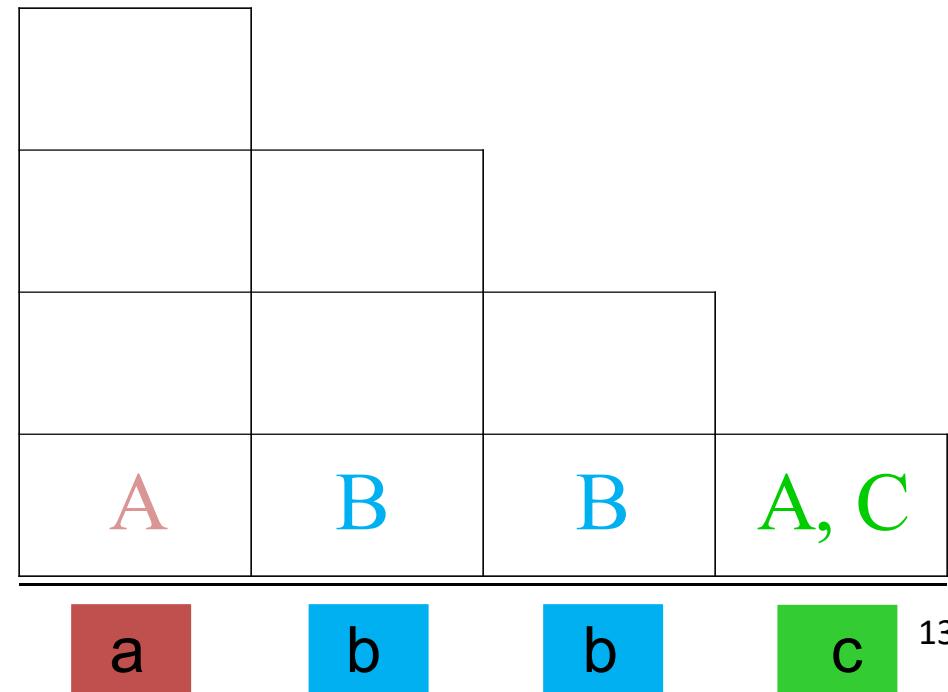
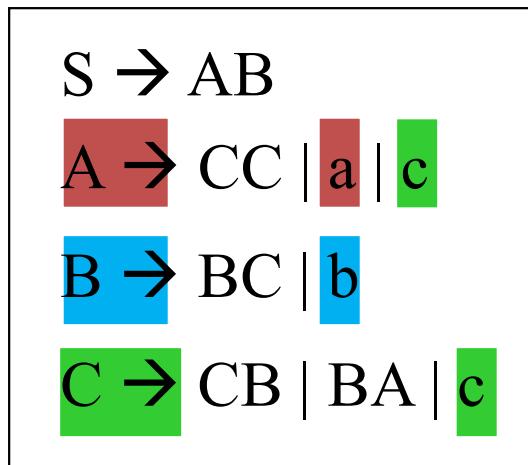
# Xây dựng bảng CYK

- Ký hiệu  $sub(i, j)$  là xâu con của xâu đang xét độ dài  $j$  tính từ ký hiệu thứ  $i$ .
- Mỗi ô chứa các ký hiệu không kết thúc suy dẫn ra  $sub(i, j)$



# Ví dụ mô phỏng

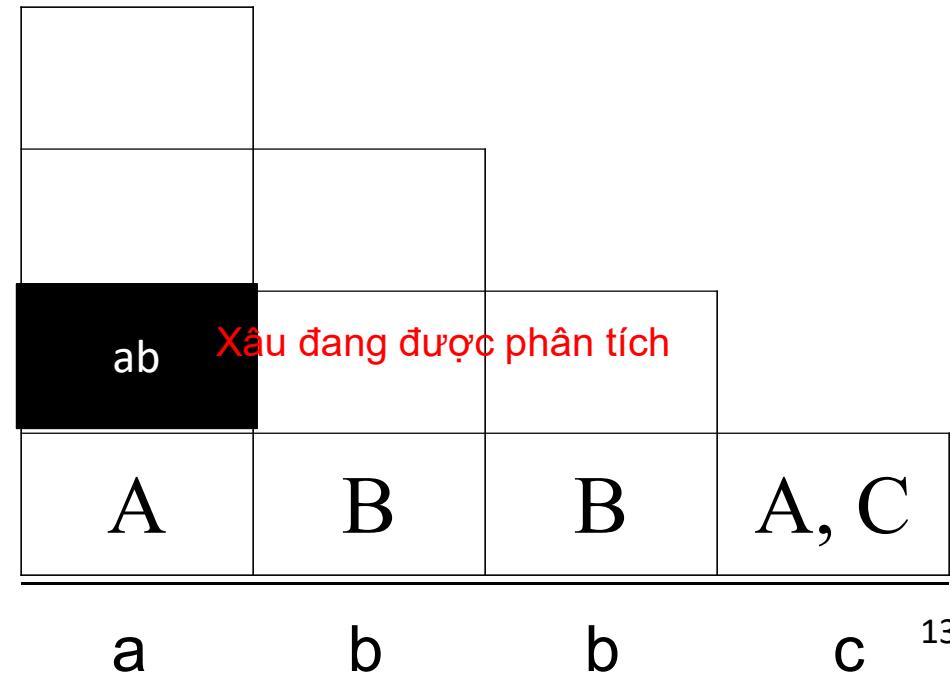
- Cơ sở: xâu con độ dài = 1
  - Các ký hiệu không kết thúc được chọn bằng cách duyệt qua sản xuất.



# Mô phỏng giải thuật CYK

- Lắp : độ dài = 2
  - Với mỗi xâu con độ dài 2
    - Phân tách thành các xâu con ngắn hơn
    - Duyệt các ô dưới nó để tính ra ký hiệu không kết thúc cần đưa vào ô.

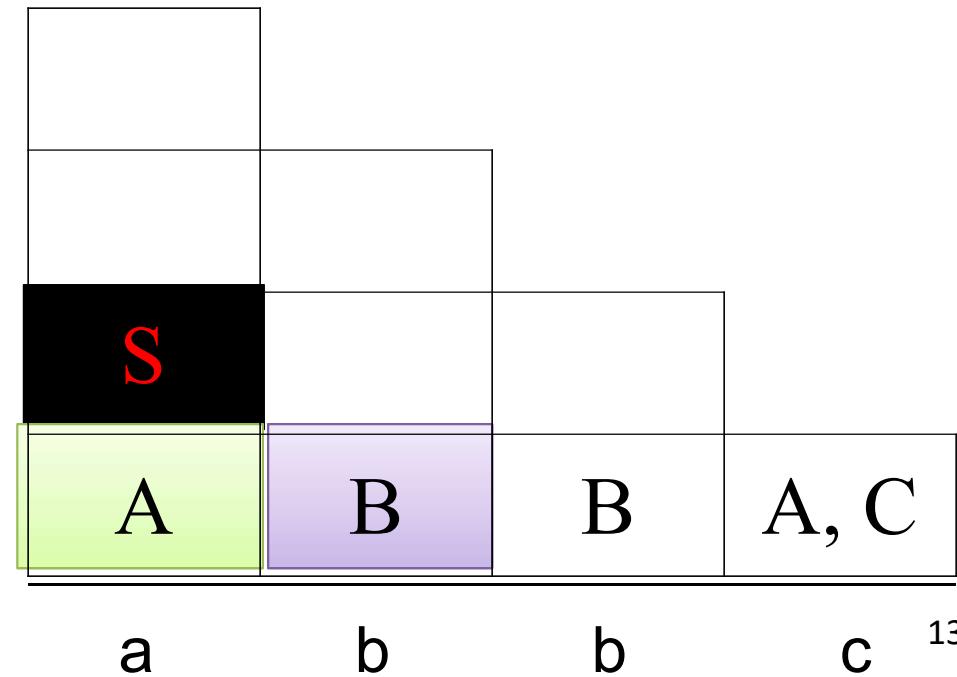
$S \rightarrow AB$   
 $A \rightarrow CC \mid a \mid c$   
 $B \rightarrow BC \mid b$   
 $C \rightarrow CB \mid BA \mid c$



# Mô phỏng giải thuật CYK

- $Xâu sub(1,2) = ab$ , có thể phân tách thành:
  - $ab = a \cdot b$   
 $= sub(1,1) + sub(2,1)$
  - Các lựa chọn: AB
  - Tìm được : S

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$



# Mô phỏng giải thuật CYK

- $sub(2,2) = bb$ , có thể chia thành:

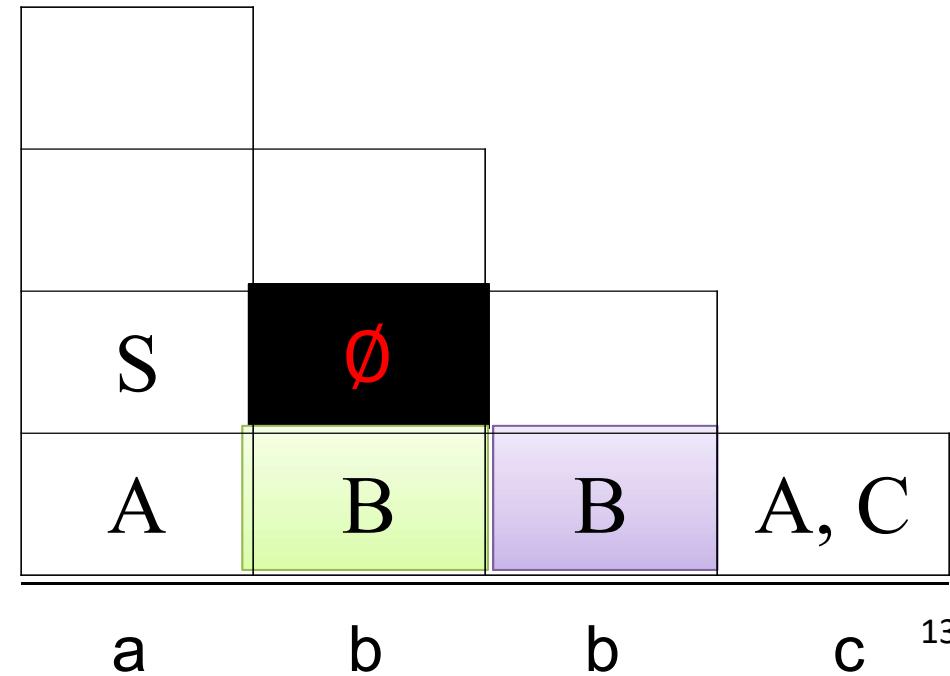
- $bb = b \ b$   
 $= sub(2,1) + sub(3,1)$

- Có thể chọn VP: BB

- Không có SX :  $\emptyset$

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$

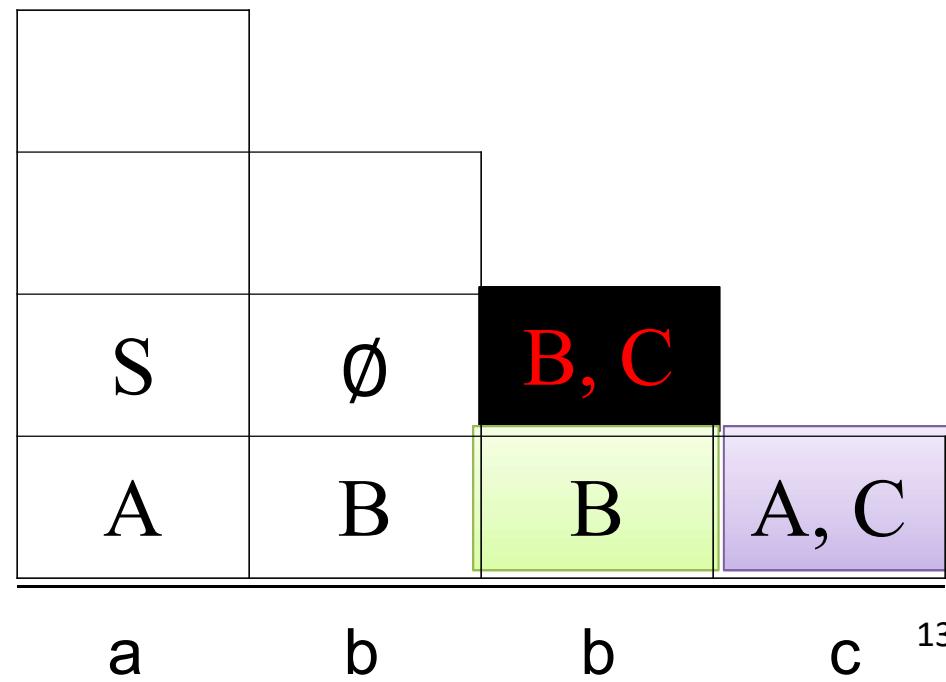
Không tìm được sản xuất nào  
→ Dòng này VP G không sinh  
được



# Mô phỏng giải thuật CYK

- $sub(3,4) = bc$ , phân tách thành
  - $bc = b$  và  $c$   
 $= sub(3,1) + sub(4,1)$
  - Các vé phải: BA, BC
  - Vé trái : B, C

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$



# Mô phỏng giải thuật CYK

- $sub(1,3) = abb$ :

- $abb = ab$  và  $b$

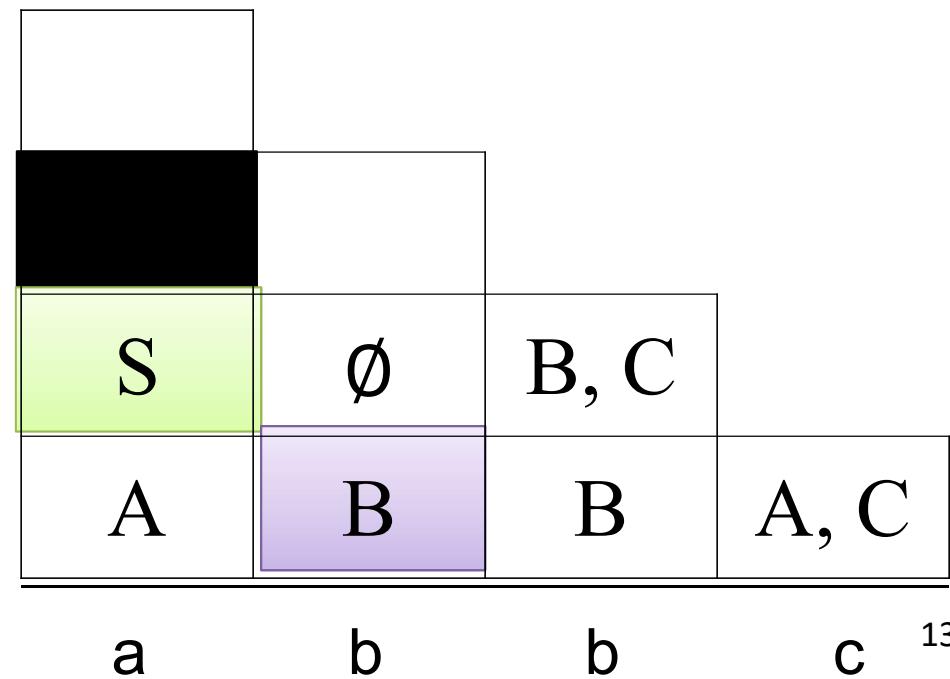
$$= sub(1,2) + sub(3,1)$$

Không có vé trái nào phù hợp cho cách phân tách này. Thủ cách khác.

- Vé phải: SB

- Vé trái :  $\emptyset$

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$



# Mô phỏng giải thuật CYK

- $sub(1,3) = abb:$

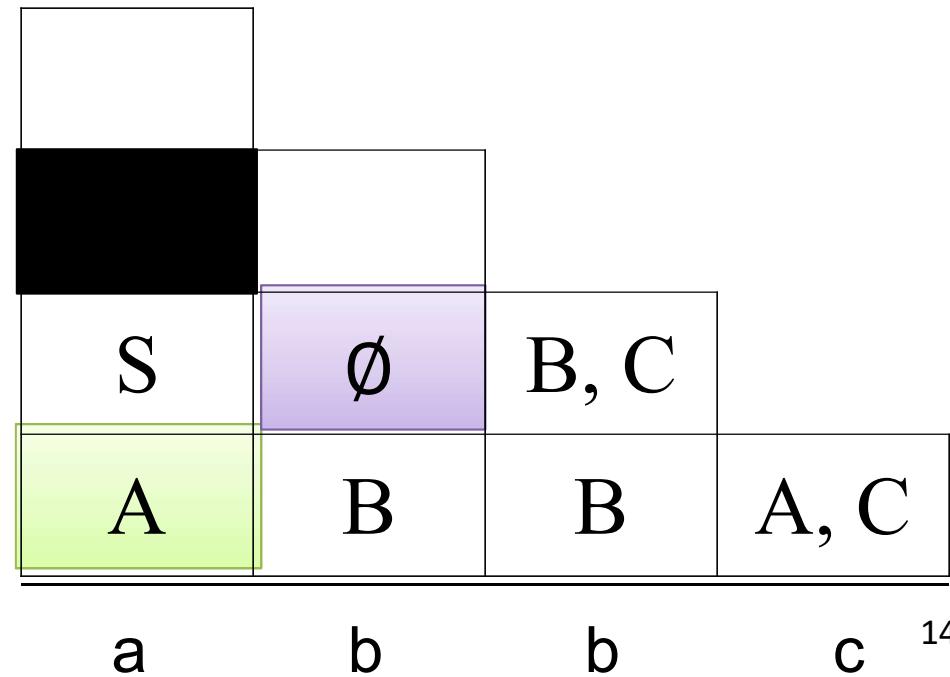
- $abb = a$  và  $bb$

$$= sub(1,1) + sub(2,2)$$

- Vé phải:  $\emptyset$

Không thể phân tích các dòng nhỏ hơn  $\rightarrow$  Không có phân tích cho xâu này  $\rightarrow$  Không cần tìm về trái.

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$



# Mô phỏng giải thuật CYK

- $sub(1,3) = \text{abb}$ :
  - $\text{abb} = sub(1,2) sub(2,1)$  không có phân tích
  - $\text{abb} = sub(1,2) sub(3,3)$  không có phân tích
- Không thể phân tích

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$

$\emptyset$			
S	$\emptyset$	B, C	
A	B	B	A, C

a

b

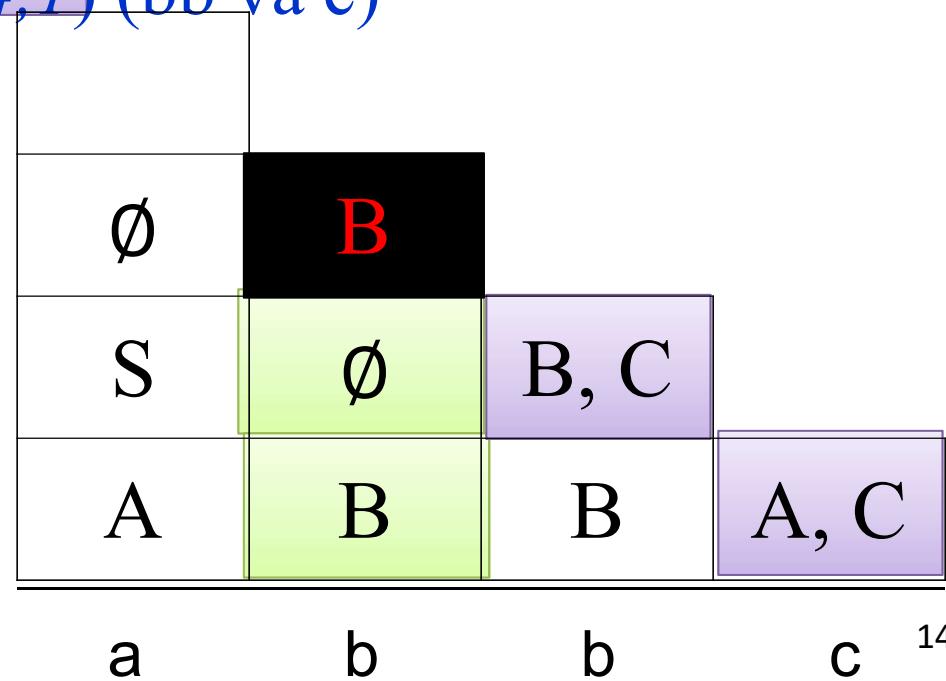
b

c

# Mô phỏng giải thuật CYK

- $sub(2,3) = bbc$ :
  - $bbc = sub(2,1) + sub(3,2)$  (b và bc)
    - Về phải: BB, BC
    - Về trái: B
  - $bbc = sub(2,2) + sub(4,1)$  (bb và c)
    - Về phải:  $\emptyset$

$S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$

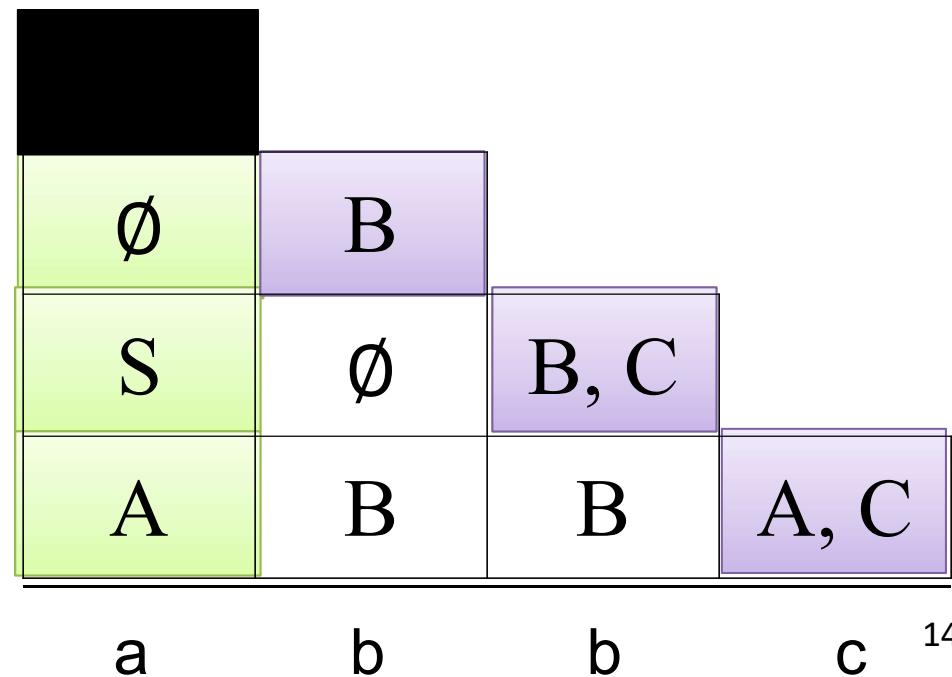


# Mô phỏng giải thuật CYK

- $sub(1,4) = abbc:$ 
  - Các vế phải : AB, SB, SC
  - Vế trái: S

Ô ở hàng trên cùng chứa S  
→ abbc đúng cú pháp

**S → AB**  
A → CC | a | c  
B → BC | b  
C → CB | BA | c

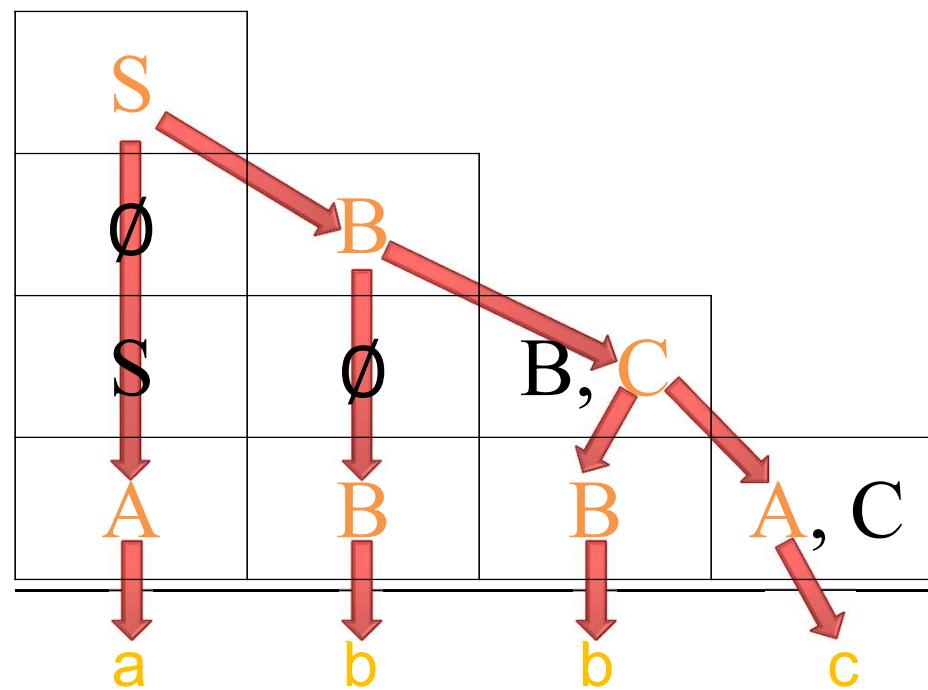


# Dụng cây phân tích cú pháp

- $abbc$  đúng cú pháp!
- Dụng cây cú pháp (phân tích trái)
  - Lần theo các suy dẫn:
    - $sub(1,4)$  được suy dẫn từ  $sub(1,1)$  và  $sub(2,3)$  áp dụng sản xuất  $S \rightarrow AB$  (1)
    - $sub(1,1)$  sử dụng  $A \rightarrow a$  (3)
    - $sub(2,3)$  suy dẫn từ from  $sub(2,2)$  và  $sub(3,2)$  áp dụng sản xuất  $B \rightarrow BC$  (6)
    - ...
- Phân tích trái là 1356869!

# Dụng cây phân tích cú pháp

- Dụng cây từ bảng được xây dựng



# Văn phạm ngôn ngữ tự nhiên

$S \rightarrow NP\ VP$

$NP \rightarrow P$

$NP \rightarrow NA$

$VP \rightarrow V\ NP$

$N \rightarrow \text{mèo} \mid \text{chuột}$

$V \rightarrow \text{bắt}$

$A \rightarrow \text{nhỏ}$

# Giải thuật Earley

- Hình thức phân tích trên xuống
- Kết quả: phân tích phải
  - Xây dựng đồ thị giữa các ký hiệu kết thúc trong câu bằng cách lưu giữ dấu vết các trạng thái của phân tích tại điểm đang xét
  - Với mỗi vị trí từ, đồ thị chứa một tập trạng thái biểu diễn mọi phân tích bộ phận cho đến thời điểm đang xét.  $I_0$  chứa mọi phân tích bộ phận được sinh ra vào đầu câu

# Giải thuật Earley

- Khi quét toàn bộ xâu, đồ thị có
  - Số bảng  $n+1$ ;  $n$  là độ dài xâu
  - Các bảng gồm  $I_0, \dots, I_n$
  - Với mỗi vị trí của ký hiệu kết thúc, đồ thị chứa tập trạng thái biểu diễn tất cả các cây phân tích bộ phận tính đến ký hiệu hiện tại. Như vậy  $I_0$  chứa mọi cây phân tích bộ phận được sinh ra đến vị trí đầu của xâu.
  - Mỗi lối vào của bảng thể hiện
    - Các thành phần đã phân tích xong và vị trí của chúng
    - Các thành phần đang phân tích
    - Các thành phần xem trước

# Trạng thái

- Các dòng của bảng được gọi là trạng thái và biểu diễn bằng các sản xuất có chứa dấu chấm •
- Ví dụ: với văn phạm

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

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

$$F \rightarrow (E) \mid a$$

các trạng thái sau có nghĩa là

$E \rightarrow \bullet T + E$  Xem trước T

$E \rightarrow T \bullet + E$  Đang phân tích E

$E \rightarrow T \bullet$  Đã tìm <sup>149</sup> được E

# Giải thuật tạo danh sách phân tích Earley

Vào : Văn phạm phi ngữ cảnh  $G=(\Sigma, \Delta, P, S)$  ,  $w = a_1 \dots a_n \in \Sigma^*$

Ra : Danh sách phân tích  $I_0, \dots, I_n$

*Phương pháp:*

## I. Xây dựng $I_0$

1. Nếu  $S \rightarrow \alpha \in P$  thêm trạm  $[S \rightarrow .\alpha, 0]$  vào  $I_0$ . Thực hiện bước 2 và 3 đến khi không thêm được trạm mới vào  $I_0$
2. Nếu  $[B \rightarrow \gamma., 0]$  trong  $I_0$ , thêm  $[A \rightarrow \alpha B \beta, 0]$  vào  $I_0$  nếu  $[A \rightarrow \alpha B \beta, 0] \in I_0$
3. Giả sử  $[A \rightarrow \alpha B \beta, 0]$  , với mọi sản xuất  $B \rightarrow \gamma$  thêm vào  $I_0$  trạm  $[B \rightarrow .\gamma, 0]$  nếu trạm đó chưa có trong  $I_0$

## II. Xây dựng $I_j$

Giả sử đã xây dựng  $I_0, \dots, I_{j-1}$

Nếu  $[B \rightarrow \alpha.a\beta, i] \in I_{j-1}$  và  $a = a_j$  thì thêm  $[B \rightarrow \alpha a \beta, i]$  vào  $I_j$

Thực hiện các bước 1 và 2 sau đây đến khi không thêm được trạm mới vào  $I_j$

1. Giả sử  $[A \rightarrow \alpha., i] \in I_j$ . Nếu tồn tại trạm  $[B \rightarrow \alpha.A\beta, k]$  trong  $I_i$  thì thêm  $[B \rightarrow \alpha A \beta, k]$  vào  $I_j$
2. Giả sử  $[A \rightarrow \alpha.B\beta, i] \in I_j$ . Với mọi sản xuất  $B \rightarrow \gamma \in P$  thêm  $[B \rightarrow .\gamma, j]$  vào  $I_j$

- III.  $w \in L(G) \Leftrightarrow$  tồn tại  $[S \rightarrow \alpha., 0]$  thuộc  $I_n$

$$_0 a_1 + _2 a_3$$

$$[E \rightarrow \bullet \cdot T + E, 0], \in I_0$$

- 0 thứ nhất nghĩa là cấu trúc E bắt đầu từ điểm đầu của xâu vào
- 0 thứ hai cho thấy vị trí dấu chấm

$$[E \rightarrow T \bullet + E, 2] \in I_3$$

- E bắt đầu từ vị trí 2
- Dấu chấm ở vị trí 3
- T đã phân tích thành công
- + sẽ được xét tiếp theo

# Kết thúc thành công

- Câu trả lời của bộ phân tích cú pháp có được sau khi xem xét lối vào cuối cùng của biểu đồ
- Nếu lối vào có dạng  $[S \rightarrow \alpha \bullet , 0], \in I_n$ , xâu đưa vào là đúng cú pháp.
- Đồ thị sẽ chứa mọi phân tích của xâu đưa vào chứ không chỉ những phân tích thành công.

# Mô tả hình thức giải thuật Earley

## Giả sử

$G = (\Sigma, \Delta, P, S)$  là văn phạm phi ngũ cành

$w = a_1 \dots a_n \in \Sigma^*$  là xâu cần phân tích

$[A \rightarrow X_1 X_2 \dots X_k \bullet X_{k+1} \dots X_m, i]$  là một trạng thái (state) của  $w$  nếu

- $A \rightarrow X_1 X_2 \dots X_m \in P$
- $0 \leq i \leq n$
- $\bullet$  là ký hiệu đặc biệt không thuộc  $V = \Sigma \cup \Delta$

# Xây dựng danh sách bảng

$\forall j, 0 \leq j \leq n$ , xây dựng một danh sách các bảng  $I_j$  sao cho

- $[A \rightarrow \alpha \bullet \beta, i] \in I_j$  nếu và chỉ nếu  $\exists \gamma, \delta$  sao cho
  - $S \Rightarrow^* \gamma A \delta$
  - $\gamma \Rightarrow^* a_1 \dots a_i$
  - $\alpha \Rightarrow^* a_{i+1} \dots a_j$
- Dãy  $I_0, \dots, I_n$  là danh sách phân tích đối với xâu vào  $w$ .
- $w \in L(G)$  khi và chỉ khi trạng thái  $[S \rightarrow \alpha, 0] \in I_n$

# Giải thuật tạo danh sách phân tích Earley

Vào : Văn phạm phi ngũ cành  $G=(\Sigma, \Delta, P, S)$  ,  $w = a_1 \dots a_n \in \Sigma^*$

Ra : Danh sách bảng phân tích  $I_0, \dots, I_n$

*Phương pháp: Xây dựng bảng  $I_0$*

1. Nếu  $S \rightarrow \alpha \in P$  thêm trạng thái  $[S \rightarrow .\alpha, 0]$  vào  $I_0$ . Thực hiện bước 2 và 3 đến khi không thêm được trạng thái mới vào  $I_0$
2. Nếu  $[B \rightarrow \gamma., 0]$  trong  $I_0$ , thêm  $[A \rightarrow \alpha B \beta, 0]$  vào  $I_0$  nếu  $[A \rightarrow \alpha B \beta, 0] \in I_0$
3. Giả sử  $[A \rightarrow \alpha B \beta, 0]$  , với mọi sản xuất  $B \rightarrow \gamma$  thêm vào  $I_0$  trạm  $[B \rightarrow .\gamma, 0]$  nếu trạm đó chưa có trong  $I_0$

Xây dựng bảng  $I_j$       Giả sử đã xây dựng  $I_0, \dots, I_{j-1}$

4. Nếu  $[B \rightarrow \alpha.a\beta, i] \in I_{j-1}$  và  $a = a_j$  thì thêm  $[B \rightarrow \alpha a \beta, i]$  vào  $I_j$ . Thực hiện các bước 5 và 6 đến khi không thêm được trạng thái mới vào  $I_j$
5. Giả sử  $[A \rightarrow \alpha., i] \in I_j$ . Nếu tồn tại trạng thái  $[B \rightarrow \alpha.A\beta, k]$  trong  $I_i$  thì thêm  $[B \rightarrow \alpha A \beta, k]$  vào  $I_j$
6. Giả sử  $[A \rightarrow \alpha.B\beta, i] \in I_j$ . Với mọi sản xuất  $B \rightarrow \gamma \in P$  thêm  $[B \rightarrow .\gamma, j]$  vào  $I_j$

**I<sub>0</sub>**

[E → .T+E, 0], [E → .T, 0]

[T → .F\*T, 0], [T → .F, 0]

[F → .(E), 0], [F → .a, 0]

**I<sub>2</sub> (a<sub>2</sub> = +)**

[E → T+.E, 0],

[E → .T+E, 2], [E → .T, 2]

[T → .F\*T, 2], [T → .F, 2]

[F → .(E), 2], [F → .a, 2]

**I<sub>1</sub> (a<sub>1</sub> = a)**

[F → a., 0]

[T → F.\*T, 0], [T → F., 0]

[E → T.+E, 0], [E → T., 0]

**I<sub>3</sub> (a<sub>3</sub> = a)**

[F → a., 2]

[T → F.\*T, 2], [T → F., 2]

[E → T.+E, 2], [E → T., 2]

[E → T+E., 0]

# Giải thuật tìm phân tích phải

Vào:

- Văn phạm phi ngữ cảnh  $G = (\Sigma, \Delta, P, S)$  không chứa chu trình, không chứa  $\varepsilon$ - sản xuất. Các sản xuất của  $G$  được đánh số 1 - p
- Xâu  $w = a_1 \dots a_n$
- Danh sách phân tích  $I_0, I_1, \dots, I_n$

Ra:

Phân tích phải p của w hoặc thông báo sai

# Giải thuật tìm phân tích phải

## *Phương pháp*

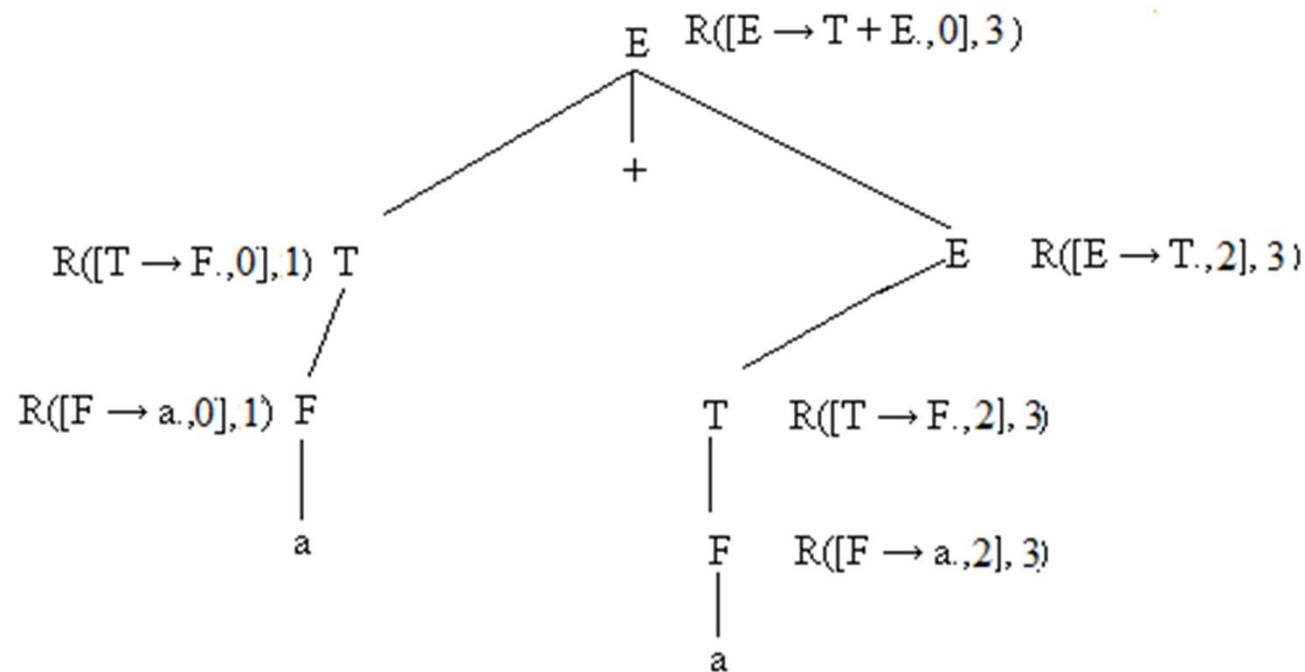
Nếu không tồn tại trạng thái  $[S \rightarrow \alpha., 0]$  trong  $I_n$  thì dừng và thông báo sai.

Ngược lại, xây dựng phân tích phải  $\pi.\pi$  được lưu trữ trong một biến toàn cục.

- Khởi tạo biến  $\pi$ ,  $\pi := \varepsilon$
- Thực hiện thủ tục đệ quy  $R([S \rightarrow \alpha., 0, n])$  với thủ tục  $R([A \rightarrow \beta., i], j)$  được định nghĩa như sau:
  1.  $\pi := h \pi$ ,  $h$  là số thứ tự của sản xuất  $A \rightarrow \beta$
  2. Nếu  $\beta = X_1 X_2 \dots X_m$ , đặt  $k := m$  và  $l := j$
  3. Xét các trường hợp
    - a. Nếu  $X_k \in \Sigma$ ,  $k := k-1$ ;  $l := l-1$ ;
    - b. Nếu  $X_k \in \Delta$ , tìm trạng thái  $[X_k \rightarrow \gamma., r]$  trong  $I_l$  trong đó  $[A \rightarrow X_1 X_2 \dots X_{k-1} \cdot X_k \dots X_m, i] \in I_r$ .  
Thực hiện  $R([X_k \rightarrow \gamma., r], l)$ ;  
 $k := k-1$ ;  
 $l := r$

Lặp lại bước 3 đến khi  $k = 0$

# Minh họa cách tìm phân tích phải



## 4.3. Phân tích topdown tiền định

**FIRST VÀ FOLLOW**

**Văn phạm LL(k)**

**Văn phạm LL(1)**

**Phân tích LL(1)**

**Phương pháp đệ quy trên xuống**

# Các phương pháp tiền định để phân tích cú pháp

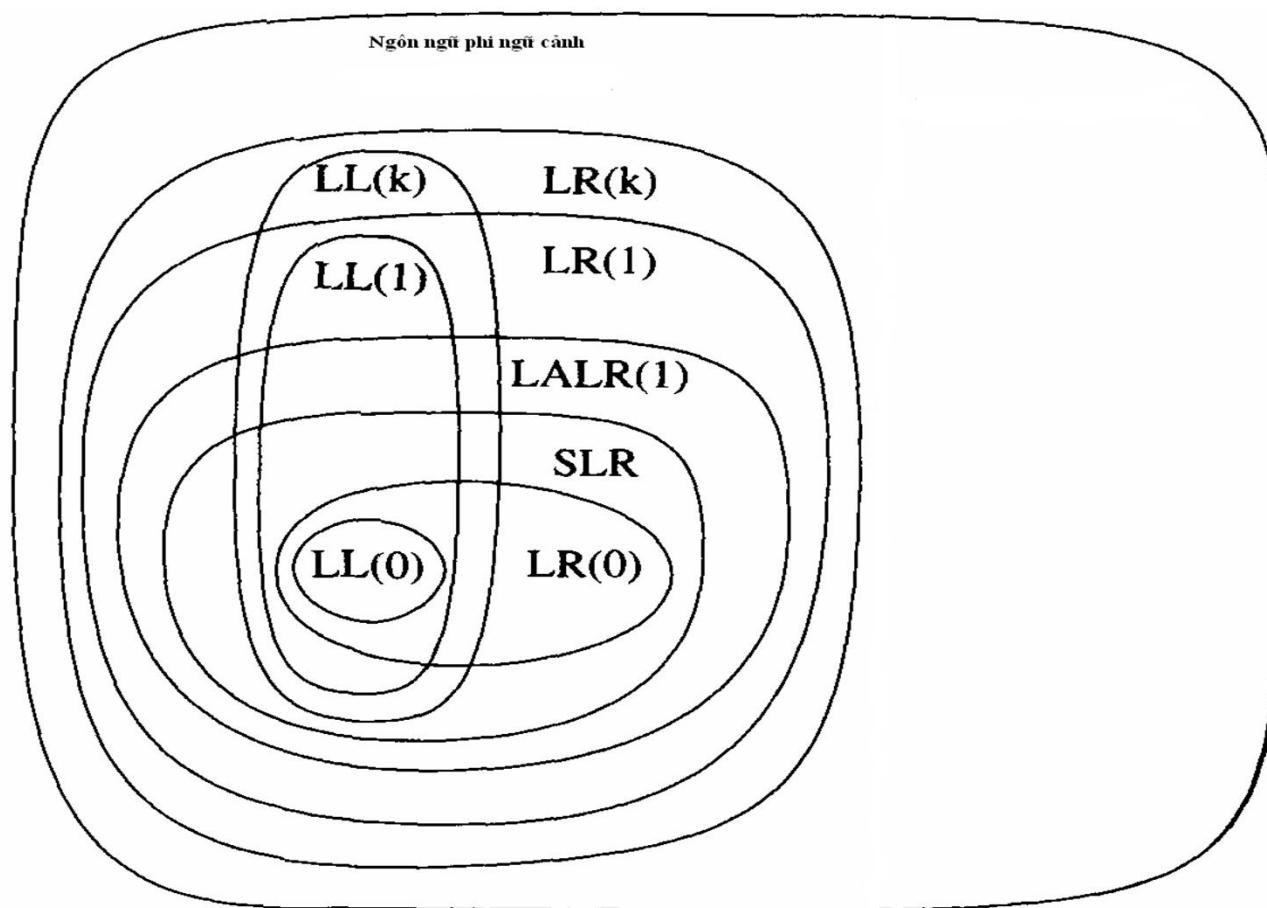
3.1. Văn phạm LL(k)

3.2. Phân tích LL(1)

3.3. Phương pháp đệ quy trên xuống

3.4. Phân tích LR

## 3.1. Văn phạm LL(k)



# Ngôn ngữ LL(k)

- Xem trước k ký hiệu trên xâu vào để quyết định sản xuất được sử dụng
- Được sinh ra nhờ văn phạm LL(k)

## FIRST<sub>k</sub>(α)

Định nghĩa : Cho văn phạm G phi ngữ cảnh, số nguyên dương k , α là một xâu bao gồm ký hiệu kết thúc và không kết thúc

*FIRST<sub>k</sub>(α) là tập các xâu x gồm k ký hiệu kết thúc trái nhất của các xâu suy dẫn từ α (Kể cả trường hợp x không có đủ k ký hiệu nhưng α suy dẫn ra x , không còn ký hiệu nào sau x)*

## FIRST<sub>k</sub>(α)

Định nghĩa : Cho văn phạm  $G = (\Sigma, \Delta, P, S)$ , số nguyên

dương  $k$  ,  $\alpha \in V^*$

$$\text{FIRST}_k(\alpha) = \{ x \in \Sigma^* \mid \alpha \Rightarrow^* x\beta \text{ và } |x| = k \text{ hoặc } \alpha \Rightarrow^* x \text{ và } |x| < k \}$$

( Tập các xâu  $x \in \Sigma^*$  có  $k$  ký hiệu trái nhất suy dẫn từ  $\alpha$  ( Kể cả trường hợp  $x$  không có đủ  $k$  ký hiệu nhưng  $\alpha x$  , không còn ký hiệu nào sau  $x$ ))

## Ví dụ FIRST<sub>k</sub>(α)

- Cho văn phạm

1.  $E \rightarrow T+E$
2.  $E \rightarrow T$
3.  $T \rightarrow F^* T$
4.  $T \rightarrow F$
5.  $F \rightarrow (E)$
6.  $F \rightarrow a$

$\alpha = T+E$

$\text{FIRST}_1(\alpha) = \{(, a\}$

$\text{FIRST}_2(\alpha) = \{(a, (, a^*, a^+\}$

$\alpha = F$

$\text{FIRST}_1(\alpha) = \{(, a\}$

$\text{FIRST}_2(\alpha) = \{a, (a, (,$

## FOLLOW<sub>k</sub>(α)

*k ký hiệu kết thúc đầu tiên tiếp sau xâu được suy dẫn từ α.*

Đặc biệt , khi A là ký hiệu không kết thúc, S suy dẫn ra  $\beta A$  thì FOLLOW<sub>1</sub>(A)  $\ni \epsilon$  hoặc FOLLOW<sub>1</sub>(A)  $\ni \$$  ( $\$$  là EOF)

## FOLLOW<sub>k</sub>(α)

$$\text{FOLLOW}_k(\alpha) = \{x \in \Sigma^* \mid S \xrightarrow{*} \beta\alpha\delta \text{ và } x \in \text{FIRST}_k(\delta)\}$$

Đặc biệt, khi  $\alpha = A \in \Delta^*$ ,  $S \xrightarrow{*} \beta A$  thì  
 $\text{FOLLOW}_1(A) = \{\varepsilon\}$  (hoặc  $\text{FOLLOW}_1(A) = \{\$\}$ )

## Văn phạm LL(k)

Định nghĩa văn phạm phi ngũ cảnh  $G = (\Sigma, \Delta, P, S)$  là LL(k) với k cho trước nếu với mọi cặp suy dẫn trái

$$S \Rightarrow^* xA\alpha \Rightarrow x\beta_1\alpha \Rightarrow^* xZ_1$$

$$S \Rightarrow^* xA\alpha \Rightarrow x\beta_2\alpha \Rightarrow^* xZ_2$$

Nếu  $\text{FIRST}_k(Z_1) = \text{FIRST}_k(Z_2)$  thì  $\beta_1 = \beta_2$

## Ví dụ

Văn phạm G với các sản xuất :

$$S \rightarrow aAS \mid b$$

$$A \rightarrow bSA \mid a$$

là LL(1)

## Văn phạm LL(1) đơn giản

Văn phạm  $G = (\Sigma, \Delta, P, S)$  là LL(1) đơn giản nếu  
mọi sản xuất của văn phạm có dạng

$A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \dots \mid a_n\alpha_n, a_i \in \Sigma \ 1 \leq i \leq n$

Trong đó  $a_i \neq a_j$  với  $i \neq j$

# Điều kiện nhận biết văn phạm LL(1)

- Định lý Văn phạm  $G = (\Sigma, \Delta, P, S)$  là LL(1) khi và chỉ khi mọi tập A- sản xuất trong P có dạng
- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n, n \geq 2$  thoả mãn  
 $\text{FIRST}_1(\alpha_i) \cap \text{FIRST}_1(\alpha_j) = \emptyset, i \neq j$
- Nếu  $\alpha_i \Rightarrow^* \epsilon$  thì  
 $\text{FIRST}_1(\alpha_j) \cap \text{FOLLOW}_1(A) = \emptyset, j \neq i$

# Điều kiện LL(1) trên sơ đồ cú pháp

- Ở mỗi lối rẽ, các nhánh phải bắt đầu bằng các ký hiệu khác nhau
- Nếu biểu đồ có chứa một đường rỗng thì mọi ký hiệu đứng sau ký hiệu được biểu diễn bởi sơ đồ phải khác các ký hiệu đứng đầu các nhánh của sơ đồ

# Văn phạm KPL là LL(1)?

A	FIRST(A)	FOLLOW(A)
Block	CONST, VAR, TYPE, PROCEDURE, BEGIN	.,;
Unsignedconst	ident, number, '	
Constant	+,-,',ident,number	
Type	ident, integer, char, array	
Statement	ident, CALL, BEGIN, WHILE, FOR	.,;, ELSE, END
Expression	+,-,(,ident,number	.,;, ELSE, END, TO, THEN, DO, ) , -, .), <, <=, >, >=, =, !=
Term	ident, number, (	.,;, END, TO, THEN, DO, ), - , <, <=, >, >=, =, !=
Factor	ident, number, (	.,;, END, TO, THEN, DO, +, -, *, /, ) , <, <=, >, >=, =, !=

## 3.2. Phân tích LL(1)

- Tư tưởng chính của phân tích cú pháp trên xuống :
  - 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
  - Có thể quay lui
- Có thể tránh được quay lui?
  - Cho sản xuất  $A \rightarrow \alpha \mid \beta$  bộ phân tích cú pháp cần chọn giữa  $\alpha$  và  $\beta$
- 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?

# 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  $\alpha$  là một xâu chứa ký hiệu kết thúc và không kết thúc,  $x \in \text{FIRST}(\alpha)$  nếu từ  $\alpha$  có thể suy dẫn ra  $x\gamma$  ( $x$  chứa 0 hoặc 1 ký hiệu)
- Nếu  $\text{FIRST}(\alpha)$  và  $\text{FIRST}(\beta)$  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 LL(1)

- Tính FIRST(X):
  - Nếu  $X$  là ký hiệu kết thúc  $\text{FIRST}(X)=\{X\}$
  - Nếu  $X \rightarrow \epsilon$  là một sản xuất thì thêm  $\epsilon$  vào  $\text{FIRST}(X)$
  - Nếu  $X$  là ký hiệu không kết thúc và  $X \rightarrow Y_1 Y_2 \dots Y_n$  là một sản xuất,
    - Thêm  $\text{FIRST}(Y_1)$  vào  $\text{FIRST}(X)$
    - Thêm  $\text{FIRST}(Y_{i+1})$  vào  $\text{FIRST}(X)$  nếu  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_j)$  chứa  $\epsilon$

# Phân tích tiền định

- Điều gì xảy ra nếu ta có sản xuất để chọn là  $A \rightarrow \alpha$  với  $\alpha = \varepsilon$  hoặc  $\alpha \Rightarrow^* \varepsilon$ ?
- Có thể mở rộng nếu ta biết rằng có một dạng câu mà ký hiệu đang xét xuất hiện sau A
- Định nghĩa:
  - Với A là ký hiệu không kết thúc,  $x \in FOLLOW(A)$  nếu và chỉ nếu Scó thể suy dẫn ra  $\alpha A x \beta$

## Tính FOLLOW

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

# Phân tích LL(1)

- 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, PÁSCAL-S

# 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

## Ví dụ

- Văn phạm:

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \varepsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \varepsilon \\ F \rightarrow (E) \mid id \end{array}$$

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

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

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

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

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

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

Văn phạm này có thể xây dựng bộ phân tích tiền định

# Bảng phân tích

	+	*	(	)	id	\$
E	$E' \rightarrow +TE'$					
E'					$E' \rightarrow \epsilon$	
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	
+	Đây					
*		Đây				
(			Đây			
)				Đây		
id					Đây	
#						Nhận

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

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

### 3.3. Phương pháp đệ quy trên xuống

- Sử dụng để phân tích cú pháp cho các văn phạm LL(1)
- Có thể mở rộng cho văn phạm LL(k), nhưng việc tính toán phức tạp
- Sử dụng để phân tích văn phạm khác có thể dẫn đến lặp vô hạn

# Bộ phân tích cú pháp

- Bao gồm một tập thủ tục, mỗi thủ tục ứng với một sơ đồ cú pháp (một ký hiệu không kết thúc)
- Các thủ tục đệ quy : khi triển khai một ký hiệu không kết thúc có thể gặp các ký hiệu không kết thúc khác, dẫn đến các thủ tục gọi lẫn nhau, và có thể gọi trực tiếp hoặc gián tiếp đến chính nó.

## Mô tả chức năng

- Giả sử mỗi thủ tục hướng tới một đích ứng với một ký hiệu không kết thúc
- Tại mỗi thời điểm luôn có một đích được triển khai, kiểm tra cú pháp hết một đoạn nào đó trong văn bản nguồn( vẽ được cây phân tích cú pháp đến một mức nào đó

# Thủ tục triển khai một đích

- Đổi chiểu văn bản nguồn với một đường trên sơ đồ cú pháp
- Đọc từ tố tiếp
- Đổi chiểu với nút tiếp theo trên sơ đồ
  - Nếu là nút tròn (ký hiệu kết thúc) thì từ tố vừa đọc phải phù hợp với từ tố trong nút
  - Nếu là nút chữ nhật nhãn A (ký hiệu không kết thúc), từ tố vừa đọc phải thuộc FIRST (A) => tiếp tục triển khai đích A
- Ngược lại, thông báo một lỗi cú pháp tại điểm đang xét

# Tùy sơ đồ thành thủ tục

- Mỗi sơ đồ ứng với một thủ tục
- Các nút xuất hiện tuần tự chuyển thành các câu lệnh kế tiếp nhau.
- Các điểm rẽ nhánh chuyển thành câu lệnh lựa chọn (if, case)
- Chu trình chuyển thành câu lệnh lặp (while, do while, repeat. . .)
- Nút tròn chuyển thành đoạn đổi chiều từ tố
- Nút chữ nhật chuyển thành lời gọi tới thủ tục khác

## Chú ý

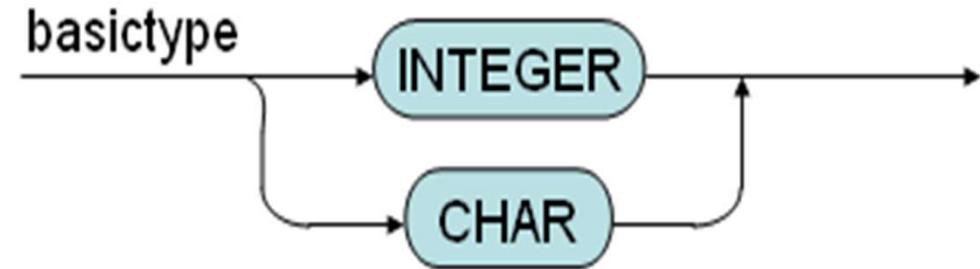
- Bộ phân tích cú pháp luôn đọc trước một từ tố
- Xem trước một từ tố cho phép chọn đúng đường đi khi gặp điểm rẽ nhánh trên sơ đồ cú pháp
- Khi thoát khỏi thủ tục triển khai một đích, có một từ tố đã được đọc dội ra
- Hàm đối chiếu từ tố: eat-> kiểm tra xem kiểu của từ tố đọc trước có phù hợp với kiểu của từ tố được sinh bởi luật không. Nếu có, đọc từ tố tiếp, ngược lại : báo lỗi

# Hàm eat - duyệt ký hiệu kết thúc

```
void eat(TokenType tokenType) {  
    if (lookAhead->tokenType == tokenType) {  
        printToken(lookAhead);  
        scan();  
    } else missingToken(tokenType, lookAhead->lineNo,  
lookAhead->colNo);  
}
```

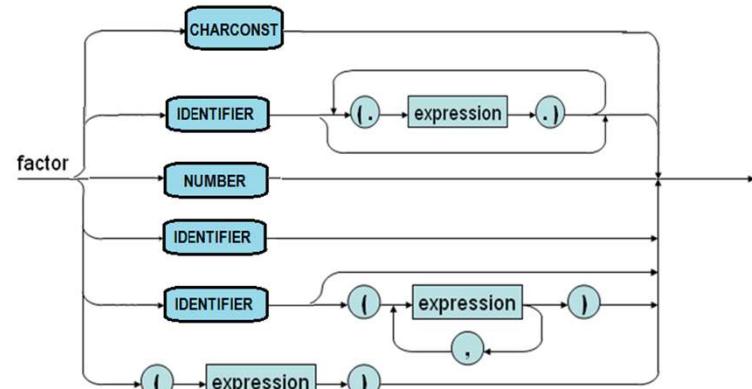
# Thủ tục CompileBasicType

```
34) BasicType ::= KW_INTEGER
35) BasicType ::= KW_CHAR
void compileBasicType
{switch(lookahead->tokenType )
{case KW_INTEGER:
eat(KW_INTEGER);
case KW_CHAR:
eat(KW_CHAR);
default: error
} }
```



# Hàm compileFactor

```
void compileFactor(void) {  
    switch (lookAhead->tokenType) {  
        case TK_NUMBER:  
            eat(TK_NUMBER);  
            break;  
        case TK_CHAR:  
            eat(TK_CHAR);  
            break;  
        case TK_IDENT:  
            eat(TK_IDENT);  
            switch (lookAhead->tokenType) {  
                case SB_LSEL:  
                    compileIndexes();  
                    break;  
                case SB_LPAR:  
                    compileArguments();  
                    break;  
                default: break;  
            }  
    }  
}
```



```
case SB_LPAR:  
    eat(SB_LPAR);  
    compileExpression();  
    eat(SB_RPAR);  
    break;  
default:  
    error(ERR_INVALIDFACTOR, lookAhead->lineNo, lookAhead->colNo);  
}  
}
```

# Hàm compileTerm (Sử dụng BNF)

## Rules for term

82) Term ::= Factor Term2

83) Term2 ::= SB\_TIMES Factor Term2

84) Term2 ::= SB\_SLASH Factor Term2

85) Term2 ::= ε

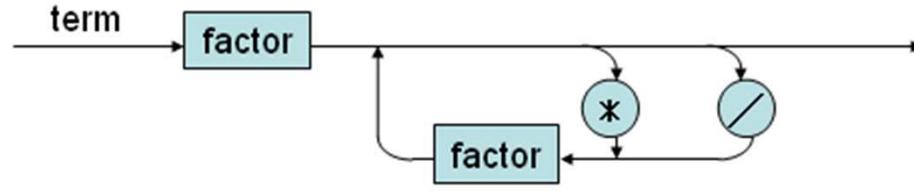
# Các hàm compileTerm và CompileTerm2 (dùng BNF)

```
void compileTerm(void) {
    compileFactor();
    compileTerm2();
}

void compileTerm2(void) {
    switch (lookAhead->tokenType) {
        case SB_TIMES:
            eat(SB_TIMES);
            compileFactor();
            compileTerm2();
            break;
        case SB_SLASH:
            eat(SB_SLASH);
            compileFactor();
            compileTerm2();
            break;
        // check the FOLLOW set
        case SB_PLUS:
        case SB_MINUS:
            case KW_TO:
            case KW_DO:
            case SB_RPAR:
            case SB_COMMAS:
            case SB_EQ:
            case SB_NEQ:
            case SB_LE:
            case SB_LT:
            case SB_GE:
            case SB_GT:
            case SB_RSEL:
            case SB_SEMICOLON:
            case KW_END:
            case KW_ELSE:
            case KW_THEN:
                break;
            default:
                error(ERR_INVALIDTERM, lookAhead->lineNo, lookAhead->colNo);
    }
}
```

# Hàm CompileTerm (dùng sơ đồ cú pháp)

```
void compileTerm(void)
{
    compileFactor();
    while(lookAhead->tokenType== SB_TIMES ||
    lookAhead->tokenType == SB_SLASH)
    {switch (lookAhead->tokenType)
    {
        case SB_TIMES:
            eat(SB_TIMES);
            compileFactor();
            break;
        case SB_SLASH:
            eat(SB_SLASH);
            break;
    }
}
```



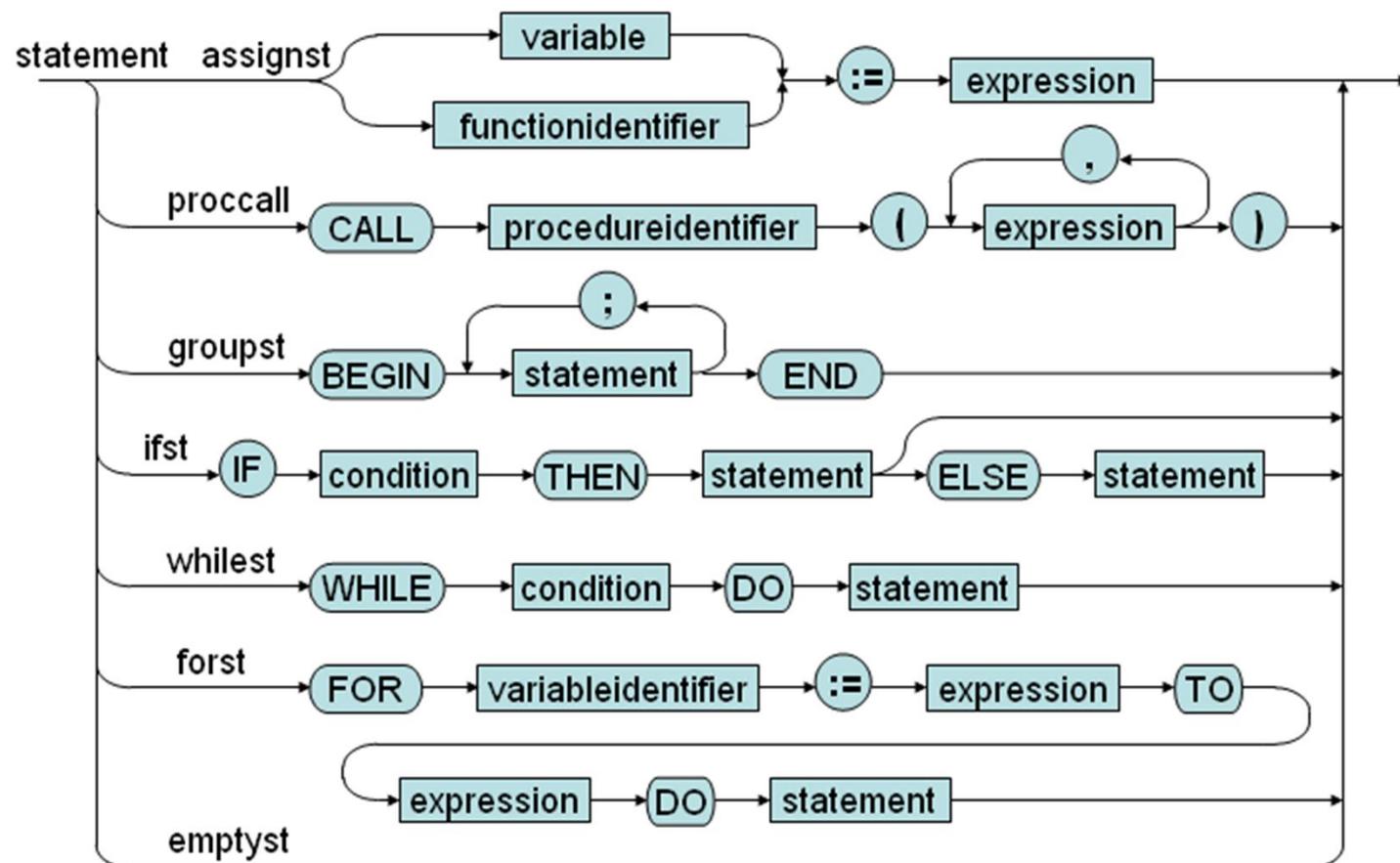
# Các luật cú pháp của lệnh

- 49) Statement ::= AssignSt
- 50) Statement ::= CallSt
- 51) Statement ::= GroupSt
- 52) Statement ::= IfSt
- 53) Statement ::= WhileSt
- 54) Statement ::= ForSt
- 55) Statement ::= ε

# Hàm compileStatement (dùng BNF)

```
void compileStatement(void) {
    switch (lookAhead->tokenType) {
        case TK_IDENT:
            compileAssignSt();
            break;
        case KW_CALL:
            compileCallSt();
            break;
        case KW_BEGIN:
            compileGroupSt();
            break;
        case KW_IF:
            compileIfSt();
            break;
        case KW WHILE:
            compileWhileSt();
            break;
        case KW FOR:
            compileForSt();
            break;
    }
    // EmptySt needs to check FOLLOW tokens
    case SB_SEMICOLON:
    case KW_END:
    case KW_ELSE:
        break;
        // Error occurs
    default:
        error(ERR_INVALIDSTATEMENT, lookAhead->lineNo, lookAhead->colNo);
        break;
    }
}
```

# Sơ đồ cú pháp cho lệnh



# Hàm compileStatement (Dùng SĐCP)

```
void compileStatement(void) {
    switch (lookAhead->tokenType) {
case TK_IDENT:
    eat(TK_IDENT);
    while (lookAhead->tokenType==SB_LSEL)
        {eat(SB_LSEL);
         compileExpression();
         eat(SB_RSEL); }
    eat(SB_ASSIGN);
    compileExpression();
    break;
case KW_CALL:
    eat(KW_CALL);
    eat(TK_IDENT);
if (lookAhead->tokenType== SB_LPAR)
    eat(SB_LPAR);
    compileExpression();
    while (lookAhead->tokenType== SB_COMMA)
        compileExpression();
    eat(SB_RPAR);
    // Check FOLLOW set .....
break;
case KW_BEGIN:.....
    break;
case KW_IF:.....
    break;
case KW WHILE:.....
    break;
case KW FOR:.....
    break;
// EmptySt needs to check FOLLOW tokens
case SB_SEMICOLON:
case KW_END:
case KW_ELSE:
    break;
    // Error occurs
default:
    error(ERR_INVALIDSTATEMENT, lookAhead->lineNo,
lookAhead->colNo);
    break;
}
}
```

# Các kỹ thuật phân tích từ dưới lên

- Phân tích cú pháp theo phương pháp gạt – thu gọn.
  - *Gạt* các ký hiệu vào cho đến khi tìm được cán.
  - Sau đó *thu gọn* xâu thành ký hiệu không kết thúc ở vé trái của sản xuất tương ứng.
- Phân tích cú pháp theo phương pháp thứ bậc toán tử
  - Dựa trên phương pháp gạt – thu gọn
  - Chỉ ra cán dựa trên các luật về thứ bậc toán tử.

# Ví dụ: Phân tích gat – thu gọn

Văn phạm

1.  $S \rightarrow E$
2.  $E \rightarrow E + E$
3.  $E \rightarrow E * E$
4.  $E \rightarrow \text{num}$
5.  $E \rightarrow \text{id}$

Xâu vào

$\text{id}_1 + \text{num} * \text{id}_2$

Cán là các đối tượng  
được gạch chân

STACK	HOẠT ĐỘNG
\$	Gạt
\$ <u><math>\text{id}_1</math></u>	Thu (luật 5)
\$ E	Gạt
\$ E <u>+</u>	Gạt
\$ E <u>+</u> <u><math>\text{num}</math></u>	Thu (luật 4)
\$ E <u>+</u> E	Gạt
\$ E <u>+</u> E <u>*</u>	Gạt
\$ E <u>+</u> E <u>*</u> <u><math>\text{id}_2</math></u>	Thu (luật 5)
\$ E <u>+</u> E <u>*</u> E	Thu (luật 3)
\$ <u>E</u> <u>+</u> E	Thu (luật 2)
\$ <u>E</u>	Thu (luật 1)
\$ S	Nhận

# Phân tích gạt thu gọn

- Một phân tích gạt- thu gọn có 4 hành động:
  - Gạt – Ký hiệu vào tiếp theo được gạt vào stack
  - Thu gọn – cán trên đỉnh stack
    - Thu gọn cán
    - Thay bằng vẽ trái thích hợp
  - Nhận – Dừng và thông báo thành công
  - Sai – Gọi thủ tục thông báo lỗi

# Phân tích gạt – thu gọn

- Khi nào tìm được cản?
  - Phân tích văn phạm.
  - Xây dựng bảng
  - Xem trước xâu vào
- Bộ phân tích cú pháp LR(1) nhận biết những ngôn ngữ mà chỉ cần xem trước một ký hiệu có thể biết được nên gạt hay thu gọn.
  - L : Phân tích xâu vào từ trái qua phải
  - R : Đưa ra phân tích phải
  - 1: Xem trước 1 ký hiệu

# Hoạt động của bộ phân tích cú pháp

- Sử dụng stack lưu trữ vết trạng thái
  - Trạng thái trên đỉnh stack tổng hợp thông tin phía dưới.
  - Stack chứa thông tin về những đối tượng đã phân tích qua.
- Sử dụng bảng phân tích để xác định hành động dựa trên trạng thái hiện thời và ký hiệu xem trước
- Làm thế nào xây dựng bảng phân tích?

# Các kỹ thuật phân tích LR

- LR đơn giản(SLR)
  - Đơn giản, dễ cài đặt,yếu
  - Áp dụng cho lớp LR(0)
- LR chính tắc (Canonical LR)
  - Mạnh nhất nhưng chi phí cao nhất
  - Sử dụng các mục của văn phạm LR(1)
- LALR( Lookahead LR)
  - Phương pháp trung gian giữa hai phương pháp trên
  - Dùng được cho phần lớn các ngôn ngữ lập trình
  - Có thể cài đặt khá hiệu quả.
  - Có thể xung đột
  - Sử dụng các mục của văn phạm LR(1)

# Ví dụ

$$E' \rightarrow E$$

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow \text{id}$$

$$S' \rightarrow S$$

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow * R$$

$$L \rightarrow \text{id}$$

$$R \rightarrow L$$

# Tìm cán

- Khi một bộ phân tích cú pháp gạt – thu gọn xử lý xâu vào, nó cần lưu trữ dấu vết mọi cán có thể có.
- Ví dụ, xét văn phạm sản sinh biểu thức và xâu  $x+y$ .
  - Giả sử bộ phân tích cú pháp đã xử lý  $x$  và thu gọn nó thành  $E$ . Khi ấy trạng thái hiện hành được biểu diễn bởi  $E \bullet +E$  ở đây  $\bullet$  nghĩa là
    - $E$  đã được phân tích và
    - $+E$  là đối tượng có khả năng đi sau, nếu tìm thấy, sẽ cho phân tích thành công.
  - Mục đích của ta là tìm được  $E+E\bullet$ , biểu diễn cán thực sự và sẽ cho kết quả khi thu gọn  $E \rightarrow E+E$

# Phân tích LR

- Phân tích LR làm việc bằng cách xây dựng một ô tô mat hữu hạn mà mỗi trạng thái biểu diễn những phân tích đã thực hiện và những phân tích ta muốn thực hiện trong tương
  - Trạng thái chứa các sản xuất có dấu chấm như đã trình bày ở trên.
  - Các sản xuất như vậy gọi là mục.
- Các trạng thái chứa cán (dấu chấm đi tới cực phải sản xuất) dẫn đến hành động thu gọn phụ thuộc vào ký hiệu xem trước

# Phân tích SLR

- Các bộ phân tích SLR xây dựng ôtômat với các trạng thái chứa các mục và thu gọn được quyết định dựa trên thông tin về tập FOLLOW.
- Ví dụ: Xây dựng bảng SLR cho văn phạm

$$\begin{array}{l} S' \rightarrow S \\ S \rightarrow L=R \\ S \rightarrow R \\ L \rightarrow *R \\ L \rightarrow id \\ R \rightarrow L \end{array}$$

# Phân tích SLR

- Khi bắt đầu phân tích, ta chưa phân tích xa xôi ào nào và ta mong đợi nhận được S. Điều đó được biểu diễn bởi  $S' \rightarrow \bullet S$ .
  - Chú ý rằng để phân tích S, ta cần phân tích  $L=R$  hoặc  $R$ . Điều đó được thể hiện qua  $S \rightarrow \bullet L = R$  và  $S \rightarrow \bullet R$
- **Bao đóng** của một trạng thái:
  - Nếu  $A \rightarrow a \bullet Bb$  biểu diễn trạng thái hiện hành và  $B \rightarrow \gamma$  là một sản xuất thì thêm  $B \rightarrow \bullet \gamma$  vào trạng thái.
  - $a \bullet Bb$  nghĩa là ta muốn xem xét B tiếp theo. Tuy nhiên phân tích B tương đương với phân tích  $\gamma$ , Vì vậy ta cũng xem xét  $\gamma$  tiếp theo.

# Phân tích SLR

- Sử dụng phép lấy bao đóng để xác định trạng thái chứa các mục LR(0). Trạng thái đầu tiên sẽ là:

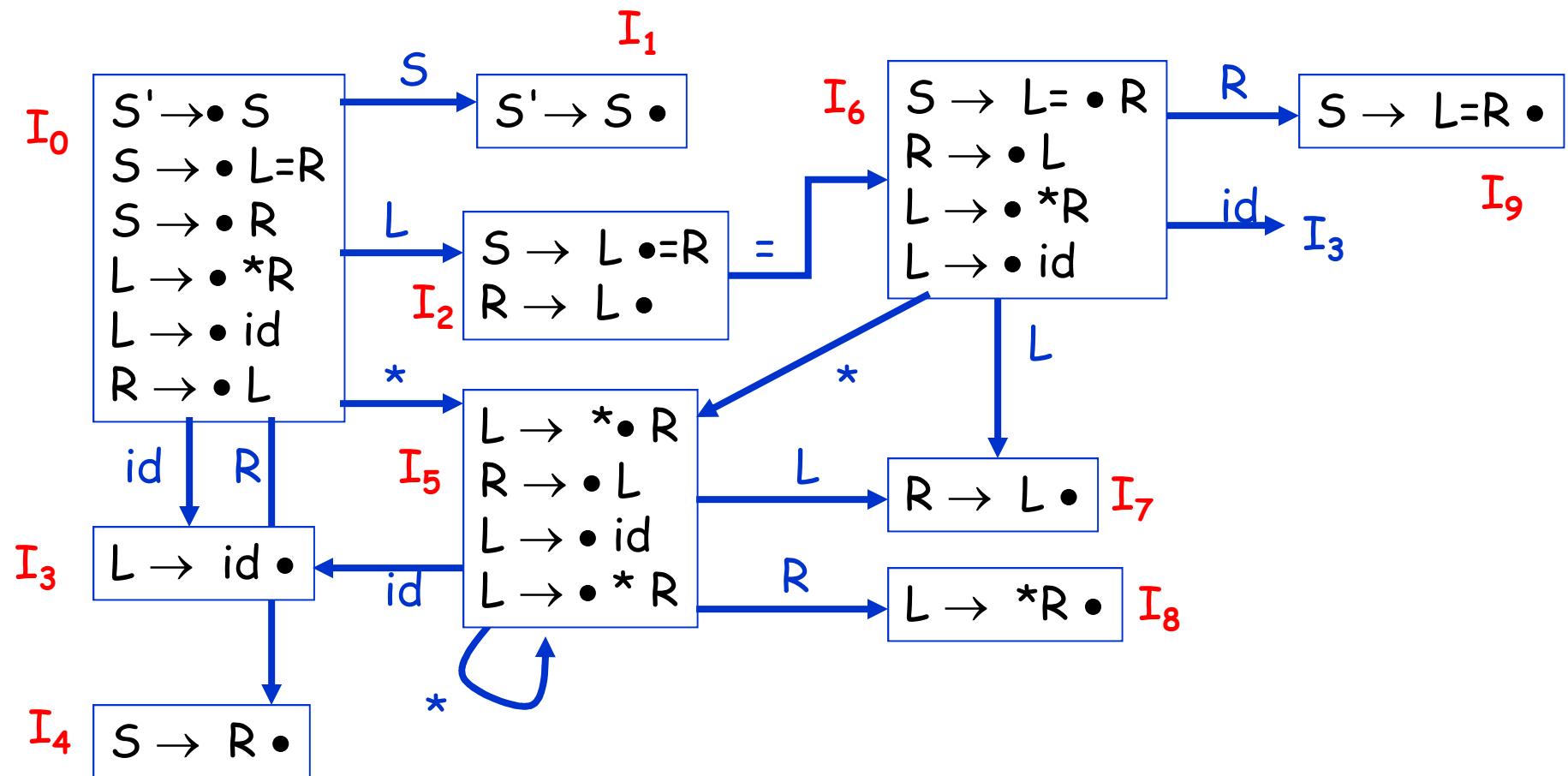
$$\begin{array}{l} S' \rightarrow \bullet S \\ S \rightarrow \bullet L=R \\ S \rightarrow \bullet R \\ L \rightarrow \bullet *R \\ L \rightarrow \bullet id \\ R \rightarrow \bullet L \end{array}$$

- Từ trạng thái này, nếu phân tích một id, ta sẽ đến trạng thái  $L \rightarrow id \bullet$
- Nếu sau một số bước, ta phân tích xâu vào và thu gọn đến L, ta đến trạng thái

$$\begin{array}{l} S \rightarrow L \bullet = R \\ R \rightarrow L \bullet \end{array}$$

# Phân tích SLR

- Tiếp tục như vậy, ta định nghĩa tất cả các trạng thái :



# Phân tích SLR

- Ô tô mat và các tập FOLLOW cho biết cách xây dựng bảng phân tích:
  - **Hành động gạt**
    - Nếu từ trạng thái  $i$  có thể đến trạng thái  $j$  khi phân tích từ tố  $t$  thì ô  $[i, t]$  của bảng sẽ chứa hành động “gạt và đến trạng thái  $j$ ”, ký hiệu là  $s_j$
  - **Hành động thu gọn**
    - Nếu trạng thái  $i$  chứa cán  $A \rightarrow \alpha \bullet$ , thì ô  $[i, t]$  của bảng phải chứa hành động “thu gọn với  $A \rightarrow \alpha$ ”, với mọi từ tố trong FOLLOW ( $A$ ), ký hiệu là  $r(A \rightarrow \alpha)$ 
      - Lý do của hành động này là ký hiệu xem trước là ký hiệu có thể đi sau  $A$ , khi ấy thu gọn  $A \rightarrow \alpha$  có thể dẫn tới phân tích đúng

# Phân tích SLR

- Ôtômat và các tập FOLLOW cho ta biết cách xây dựng bảng phân tích:
  - **Hành động thu gọn, tiếp theo**
    - Các hành động của ký hiệu không kết thúc thể hiện một số bước chuẩn bị tới hành động thu gọn.
    - Ví dụ nếu đang ở trạng thái 0 và phân tích For example, if we are in state 0 and parse a bit of input that ends up being reduced to an L, then we should go to state 2.
    - Các hành động như vậy được ghi trong một phần riêng biệt của bảng phân tích, gọi là phần GOTO

# Phân tích SLR

- Trước khi xây dựng bảng phân tích, ta cần tính toán các tập FOLLOW:

$S' \rightarrow S$
$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

$\text{FOLLOW}(S') = \{\$\}$

$\text{FOLLOW}(S) = \{\$\}$

$\text{FOLLOW}(L) = \{\$, =\}$

$\text{FOLLOW}(R) = \{\$, =\}$

# Phân tích SLR

state	action			goto			
	id	=	*	\$	S	L	R
0	s3		s5		1	2	4
1				accept			
2		s6/r( $R \rightarrow L$ )					
3		r( $L \rightarrow id$ )			r( $L \rightarrow id$ )		
4					r( $S \rightarrow R$ )		
5	s3		s5		7	8	
6	s3		s5		7	9	
7		r( $R \rightarrow L$ )			r( $R \rightarrow L$ )		
8		r( $L \rightarrow *R$ )			r( $L \rightarrow *R$ )		
9					r( $S \rightarrow L=R$ )		

Chú ý rằng quá trình gạt/thu gọn bị xung đột ở trạng thái 2 khi xem trước =

# Xung đột trong phân tích LR

- Có hai dạng xung đột trong phân tích LR:
  - Gạt/thu gọn
    - Trong một số trường hợp có thể gạt hay thu cũng được
    - Sự nhập nhằng trong lệnh if/else có thể dẫn đến tăng xung đột.
  - Thu gọn/thu gọn
    - Xung đột xảy ra khi một trạng thái chứa nhiều hơn 1 cản có thể thu gọn cho một ký hiệu xem trước.

# Xung đột trong phân tích SLR

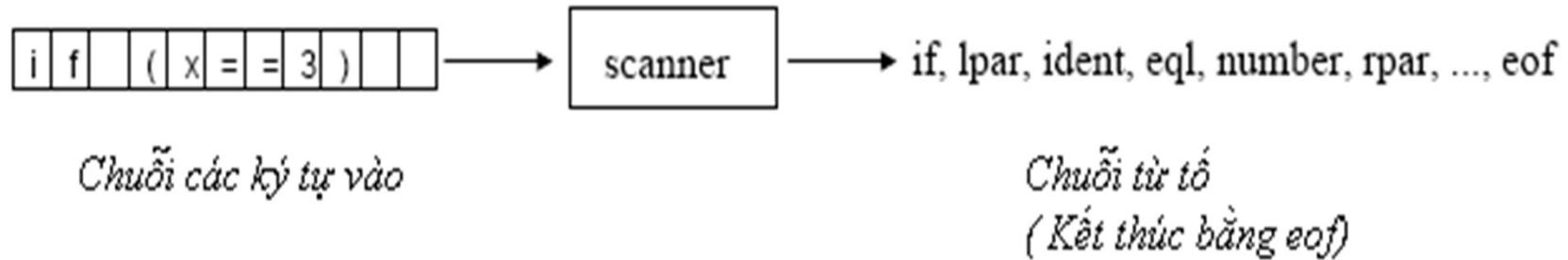
- Phát sinh xung đột trong bộ phân tích cú pháp
- Có phải do văn phạm nhập nhằng?
- Không nhất thiết, ví dụ
  - Xuất hiện xung đột khi đã phân tích L và đang xét =. Một thu gọn tại điểm này sẽ chuyển một L thành R. Tuy nhiên một thu gọn tại điểm này không phải lúc nào cũng cho phân tích đúng. Thực ra L được thu thành R khi lookahead là EOF (\$).

# Chương 4: Compiler đơn giản

- 4.1. Phân tích từ vựng
- 4.2. Phân tích cú pháp
- 4.3. Phân tích ngữ nghĩa
- 4.4. Sinh mã trung gian
- 4.5. Tối ưu mã
- 4.6. Sinh mã đích

## 4.1. Phân tích từ vựng

- Nhiệm vụ của bộ phân tích từ vựng
- Phát hiện các từ tố



- Bỏ qua các ký tự không cần thiết
  - Khoảng trắng
  - Dấu tab
  - Ký tự xuống dòng (CR,LF)
  - Chú thích

# Tùy tố có cấu trúc cú pháp

```
ident = letter {letter | digit}.
```

```
number = digit {digit}.
```

```
if = "i" "f".
```

```
eql = "=" "=".
```

```
...
```

- Tại sao không xử lý các luật này trong giai đoạn phân tích cú pháp ?

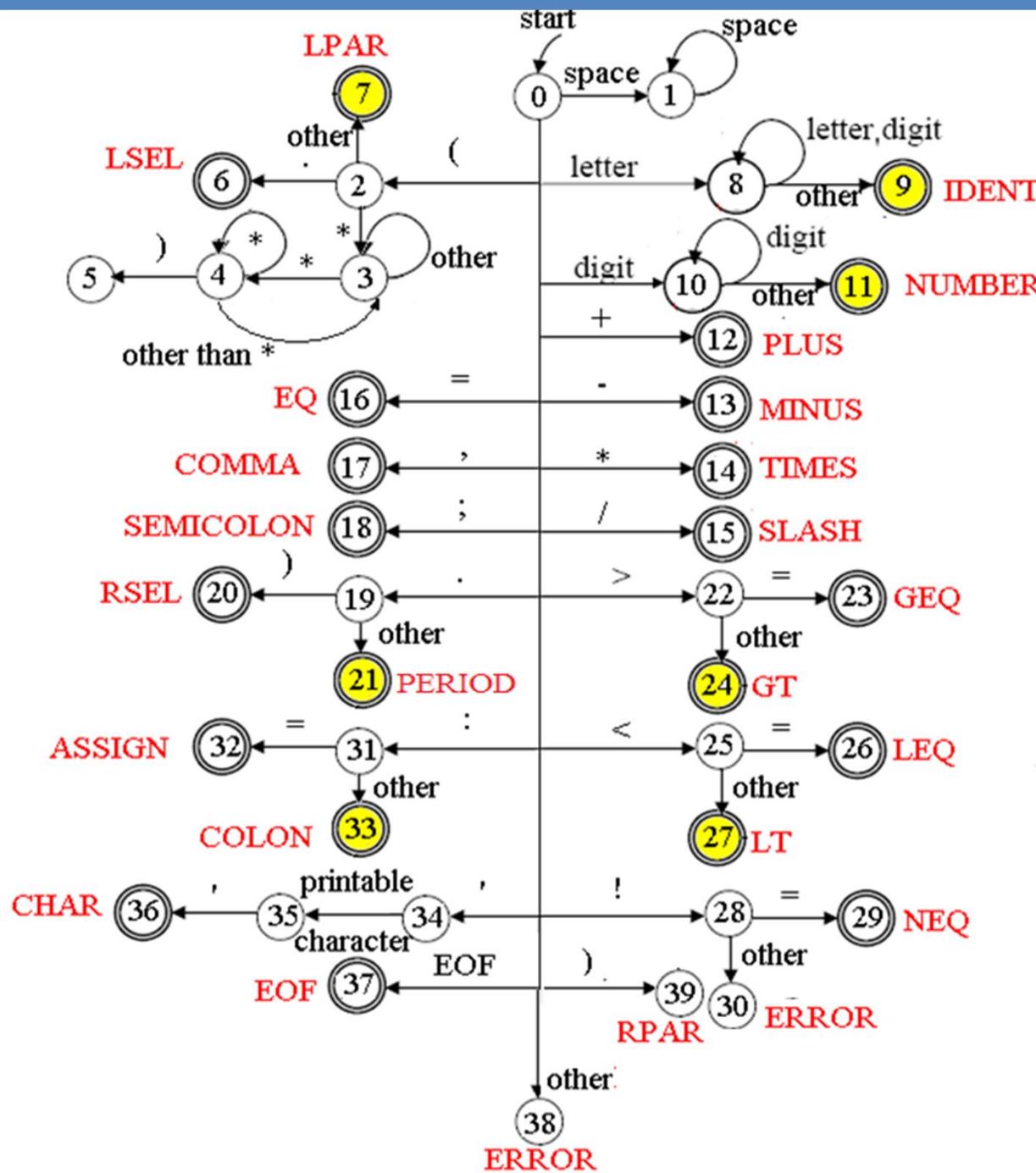
## Xử lý các luật từ vựng trong bộ phân tích cú pháp ?

- Làm cho bộ phân tích cú pháp trở nên quá phức tạp
  - Phân biệt tên và từ khoá
  - Phải có những luật phức tạp để xử lý chuỗi các ký tự không cần thiết (khoảng trắng, tab, chú thích . . .)

# Các từ tố của KPL

- Số nguyên
- Định danh
- Từ khóa: begin,end, if,then, while, do, call, const, var, procedure, program, type, function, of, integer, char, else, for, to, array
- Hằng ký tự
- Dấu phép toán:
  - số học  
+ - \* /
  - so sánh  
=      !=      <      >      <=      >=
- Dấu phân cách  
( ) . : ; ( . . )
- Dấu phép gán :=

# Ôtômat hữu hạn của bộ phân tích từ vựng KPL



Mỗi khi đoán nhận được 1 từ tố, ôtômat hữu hạn lại quay về trạng thái 0.

Với những ký tự không đoán nhận được, cần thông báo lỗi.

Nếu ô tô mat đến những trạng thái màu vàng, ký tự hiện hành đã là ký tự đầu của token tiếp theo

## Cài đặt bộ phân tích từ vựng dựa trên ô tômat

```
state = 0;  
currentChar = getCurrentChar;  
token = getToken();  
while ( token!=EOF)  
{  
    state =0;  
    token = getToken();  
}
```

# Đoán nhân từ tố

```
switch (state)
{
    case 0 : currentChar =
        getCurrentChar();
        switch (currentChar)
        {
            case space
                state = 1;
            case lpar
                state = 2;
            case letter
                state = 8;
            case digit
                state =10;
            case plus
                state = 12;
            .....
        }
}
```

# Đoán nhân tự tố (tiếp theo)

case 1:

```
while (currentChar == space) // skip blanks  
    currentChar = getCurrentChar();
```

```
state = 0;
```

case 2:

```
currentChar = getCurrentChar();
```

```
switch (currentChar)
```

```
{
```

```
case period
```

```
    state = 6; // token Isel
```

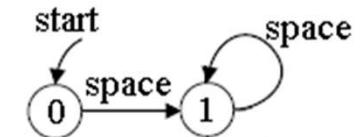
```
case times
```

```
    state = 3; // skip comment
```

```
else
```

```
    state = 7; // token Ipar
```

```
}
```



# Đoán nhân từ tố (tiếp theo)

**case 3: // skip comment**

```
currentChar = getCurrentChar();
while (currentChar != times)
{
    state = 3;
    currentChar = getCurrentChar(`-`)
```

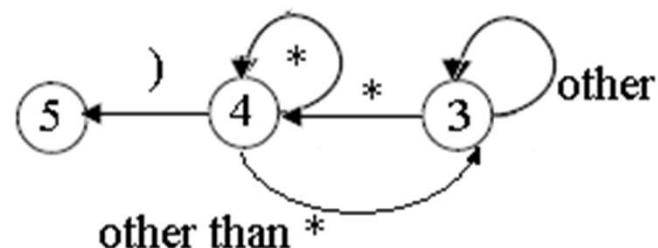
```
}
```

state = 4;

**case 4:**

```
currentChar = getCurrentChar();
while (currentChar == times)
{
    state = 4;
    currentChar = getCurrentChar();
}
```

If (currentChar == lpar) state = 5; else state =3;



## Đoán nhân tự tố (tiếp theo)

case 9:

```
if (checkKeyword (token) == TK_IDENT)  
install_ident();// save to symbol table  
else  
return checkKeyword(token);
```

.....

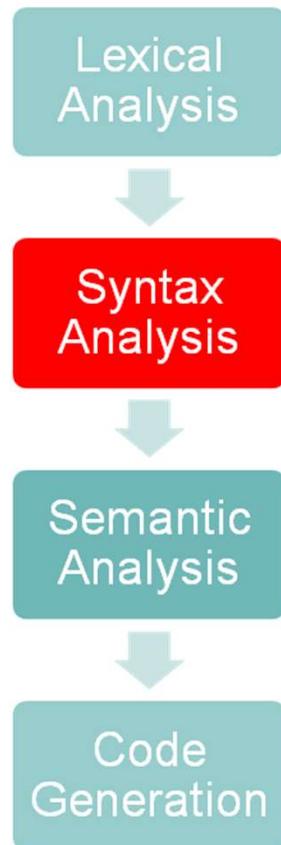
# Các thông tin trong bảng ký hiệu

- Thông tin của định danh
  - Tên: xâu ký tự
  - Thuộc tính: tên kiểu, tên biến, tên thủ tục, tên hằng...
  - Kiểu dữ liệu
  - Phạm vi sử dụng
  - Địa chỉ vùng nhớ, kích cỡ vùng nhớ
  - ...
- Với các số, thông tin về giá trị sẽ được lưu trữ

# Cấu trúc dữ liệu

```
enum {
    TK_NONE, TK_IDENT, TK_NUMBER, TK_CHAR, TK_EOF,
    KW_PROGRAM, KW_CONST, KW_TYPE, KW_VAR,
    KW_INTEGER, KW_CHAR, KW_ARRAY, KW_OF,
    KW_FUNCTION, KW_PROCEDURE,
    KW_BEGIN, KW_END, KW_CALL,
    KW_IF, KW_THEN, KW_ELSE,
    KW_WHILE, KW_DO, KW_FOR, KW_TO,
    SB_SEMICOLON, SB_COLON, SB_PERIOD, SB_COMMMA,
    SB_ASSIGN, SB_EQ, SB_NEQ, SB_LT, SB_LE, SB_GT, SB_GE,
    SB_PLUS, SB_MINUS, SB_TIMES, SB_SLASH,
    SB_LPAR, SB_RPAR, SB_LSEL, SB_RSEL
};
```

## 4.2. Phân tích cú pháp



### Nhiệm vụ của bộ phân tích cú pháp

- Kiểm tra cấu trúc ngữ pháp của một chương trình
- Kích hoạt các bộ phân tích ngữ nghĩa và sinh mã

# Xây dựng bộ phân tích cú pháp cho KPL

- Về cơ bản KPL là một ngôn ngữ LL(1)
- Thiết kế một bộ phân tích cú pháp theo phương pháp đệ quy trên xuông
  - Token ***lookAhead***
  - Duyệt ký hiệu kết thúc
  - Duyệt ký hiệu không kết thúc

## 4,3.Phân tích ngũ nghĩa

- Bài toán phân tích ngũ nghĩa
- Bảng ký hiệu và phạm vi
- Định nghĩa tựa cú pháp
- Hệ thống kiểu
- Xây dựng bộ phân tích ngũ nghĩa (bài tập)

# Bài toán phân tích ngũ nghĩa

- Kiểm tra sự tương ứng về kiểu
- Kiểm tra sự tương ứng giữa việc sử dụng hàm, biến với khai báo của chúng
- Xác định phạm vi ảnh hưởng của các biến trong chương trình
- Phân tích ngũ nghĩa thường sử dụng cây cú pháp
- Tìm ra các lỗi sau giai đoạn phân tích cú pháp
- Bước đầu phân phối bộ nhớ

## Một số vấn đề ngữ nghĩa

- Mọi định danh phải khai báo trước khi sử dụng
- Không được sử dụng các tên chuẩn cho mục đích khác
- Thiết lập hệ thống kiểu cho ngôn ngữ để kiểm tra sự tương ứng về kiểu trong toàn bộ chương trình
- Đối tượng chỉ định nghĩa một lần, phương thức của một đối tượng chỉ được

# Bảng ký hiệu và phạm vi

- Phạm vi là gì
- Quản lý phạm vi tĩnh và động
- Những vấn đề liên quan đến phạm vi
- Bảng ký hiệu
- Xây dựng bảng ký hiệu

# Quản lý phạm vi

- Đây là vấn đề liên quan đến sự phù hợp giữa khai báo và sử dụng của mỗi định danh.
- Phạm vi ảnh hưởng của mỗi định danh là phần chương trình có thể truy cập tới định danh đó.
- Một định danh có thể tham chiếu các đối tượng khác nhau trong các phạm vi khác nhau của chương trình. Phạm vi của hai định danh giống nhau không được giao nhau

# Phạm vi tĩnh và động

- Phần lớn các ngôn ngữ quản lý phạm vi theo kiểu tĩnh. Thông tin phạm vi chỉ phụ thuộc văn bản chương trình. Luật phạm vi gần nhất được áp dụng
- Một số ít ngôn ngữ cho phép quản lý phạm vi động, quản lý phạm vi khi thực hiện chương trình (Lisp, Snobol, Perl khi dùng một số từ khóa đặc biệt)
- Quản lý phạm vi động nghĩa là khi một ký hiệu được tham chiếu, chương trình dịch sẽ tham chiếu vào stack chứa các bản hoạt động để tìm ra thông tin về ký hiệu đó

# Ví dụ

## QL phạm vi tĩnh

```
1  const int b = 5;
2  int foo()
3  {
4      int a = b + 5;
5      return a;
6  }
7
8  int bar()
9  {
10     int b = 2;
11     return foo();
12 }
13
14 int main()
15 {
16     foo(); // returns 10
17     bar(); // returns 10
18     return 0;
19 }
```

## QL phạm vi động

```
1  const int b = 5;
2  int foo()
3  {
4      int a = b + 5;
5      return a;
6  }
7
8  int bar()
9  {
10     int b = 2;
11     return foo();
12 }
13
14 int main()
15 {
16     foo(); // returns 10
17     bar(); // returns 7
18     return 0;
19 }
```

# Những vấn đề về quản lý phạm vi

- Khi xét một khai báo chứa một định danh, liệu đã tồn tại định danh đó trong phạm vi hiện hành chưa?
- Khi sử dụng một định danh, liệu nó đã được khai báo chưa? Nếu nó đã được khai báo (theo luật phạm vi gần nhất) liệu khai báo có tương thích với sử dụng không?

# Bảng ký hiệu (Symbol table)

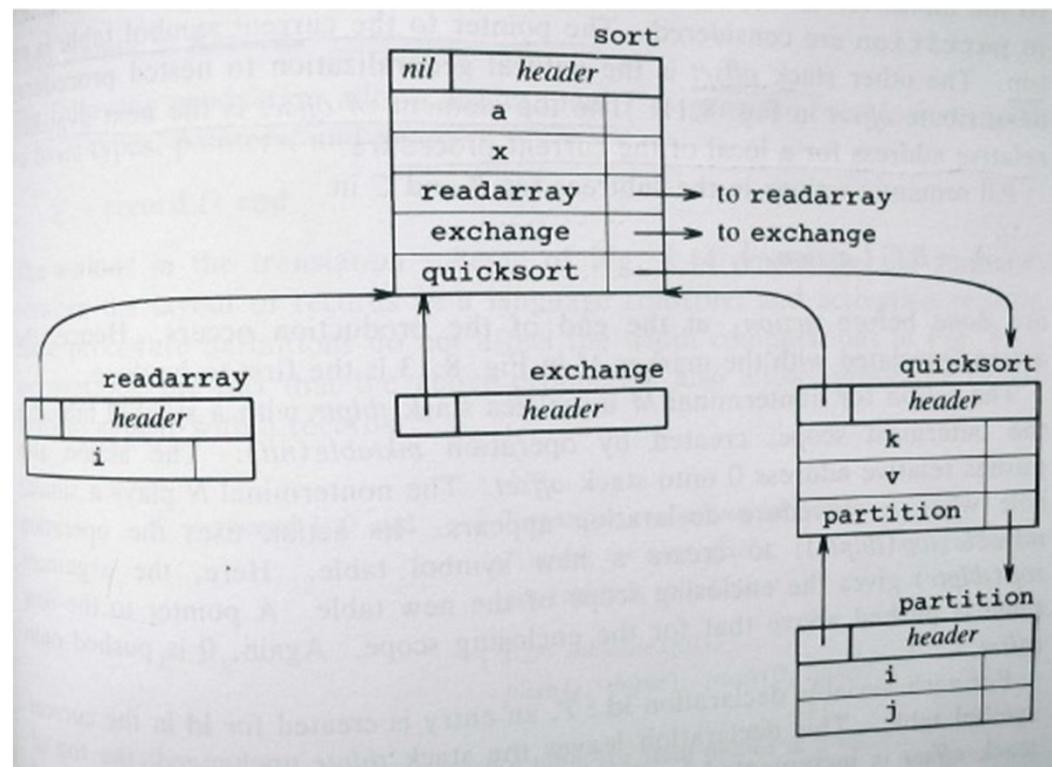
- Một cấu trúc dữ liệu cho phép theo dõi quan hệ hiện hành của các định danh (để kiểm tra ngũ nghĩa và sinh mã một cách hiệu quả)
- Phản quan trọng trong phân tích ngũ nghĩa là theo dõi các hằng/biến/kiểu/hàm/thủ tục xem có phù hợp với khai báo của chúng không?
- Khi thêm một định danh vào bảng ký hiệu, cần ghi lại thông tin trong khai báo của định danh đó.

# Ngôn ngữ có cấu trúc khối

- Khối trong ngôn ngữ lập trình là tập các cấu trúc ngôn ngữ có chứa khai báo
- Một ngôn ngữ là có cấu trúc khối nếu
  - Các khối được lồng bên trong những khối khác
  - Phạm vi của khai báo trong mỗi khối là chính khối đó và các khối chứa trong nó
- Luật lồng nhau gần nhất
  - Cho nhiều khai báo của cùng một tên. Khai báo có hiệu lực là khai báo nằm trong khối gần nhất

# Giải pháp nhiều bảng ký hiệu

- Các bảng cần được kết nối từ phạm vi trong ra phạm vi ngoài và ngược lại



# Các hàm và thủ tục trong chương trình ví dụ sort

```
program sort;
    var a : array [0..10] of integer;
        x : integer;
    procedure readarray;
        var i : integer;
        begin ... end;
    procedure exchange(i, j : integer);
        begin x := a[i]; a[i] := a[j]; a[j] := x end;
    procedure quicksort(m, n : integer);
        var k, v : integer;
        function partition(y, z : integer) : integer
            var i, j : integer;
            begin ... exchange(i, j) ... end
        begin
            if (n > m) then begin
                i := partition(m, n);
                quicksort(m, i - 1);
                quicksort(i + 1, n)
            end
        end;
    begin
        ...
        quicksort(1, 9)
    end.
```

# Xây dựng bảng ký hiệu

- Những thao tác cần thiết
- Vào phạm vi (enter scope): tạo ra một phạm vi mới trong các phạm vi lồng nhau
- Xử lý khai báo: Thêm một định danh vào bảng ký hiệu của phạm vi hiện hành
- Xử lý việc sử dụng định danh: Kiểm tra xem định danh có xuất hiện trong bảng ký hiệu của
  - Phạm vi hiện hành
  - Các phạm vi từ phạm vi hiện hành ra ngoài theo luật phạm vi gần nhất
- Ra khỏi phạm vi (exit scope): ra khỏi phạm vi hiện hành

# Các luật về phạm vi lồng nhau

- Toán tử insert vào bảng ký hiệu không được ghi đè những khai báo trước
- Toán tử lookup vào bảng ký hiệu luôn luôn tham chiếu luật phạm vi gần nhất
- Toán tử delete chỉ được xóa khai báo gần nhất

*Bảng ký hiệu hoạt động như một stack*

# Cấu trúc dữ liệu cho bảng ký hiệu

- Danh sách liên kết không sắp thứ tự:  
Thích hợp cho việc phân tích các chương trình sử dụng số lượng biến nhỏ
- Danh sách liên kết sắp thứ tự. Thao tác bổ sung tồn kém nhưng thao tác tìm kiếm lại nhanh chóng hơn
- Cây nhị phân tìm kiếm
- Bảng băm: thường được dùng nhất

# Xây dựng trong giai đoạn phân tích cú pháp

- Chỉ có thể bắt đầu nhập thông tin vào bảng ký hiệu từ khi phân tích từ vựng nếu ngôn ngữ lập trình không cho khai báo tên trùng nhau.
- Nếu cho phép các phạm vi, bộ phân tích từ vựng chỉ trả ra tên của định danh cùng với token
  - Định danh được thêm vào bảng ký hiệu khi vai trò cú pháp của định danh được phát hiện

# Định nghĩa tựa cú pháp

- Thuộc tính
- Định nghĩa tựa cú pháp
- Cây phân tích cú pháp có chú giải
- Đồ thị phụ thuộc

# Thuộc tính

- Thuộc tính là khái niệm trừu tượng biểu diễn một đại lượng bất kỳ , chẳng hạn một số, một xâu, một vị trí trong bộ nhớ . . . .
- Thuộc tính được gọi là **tổng hợp** nếu giá trị của nó tại một nút trong cây phân tích cú pháp được xác định từ giá trị của các nút con của nó.
- Thuộc tính **kế thừa** là thuộc tính tại một nút mà giá trị của nó được định nghĩa dựa vào giá trị nút cha và/hoặc các nút anh em của nó .

# Định nghĩa tựa cú pháp (syntax directed definition)

Định nghĩa tựa cú pháp là dạng tổng quát của văn phạm phi ngữ cảnh trong đó:

- Mỗi ký hiệu của văn phạm liên kết với một tập thuộc tính ,
- Mỗi sản xuất  $A \rightarrow \alpha$  liên hệ với một tập các quy tắc ngữ nghĩa để tính giá trị thuộc tính liên kết với những ký hiệu xuất hiện trong sản xuất. Tập các quy tắc ngữ nghĩa có dạng

$$b = f(c_1, c_2, \dots, c_n)$$

$f$  là một hàm và  $b$  thoả một trong hai yêu cầu sau:

- $b$  là một thuộc tính tổng hợp của  $A$  và  $c_1, \dots, c_n$  là các thuộc tính liên kết với các ký hiệu trong  $\alpha$  phải sản xuất  $A \rightarrow \alpha$
- $b$  là một thuộc tính thừa kế một trong những ký hiệu xuất hiện trong  $\alpha$ , và  $c_1, \dots, c_n$  là thuộc tính của các ký hiệu trong  $\alpha$  phải sản xuất  $A \rightarrow \alpha$

# Ví dụ một định nghĩa tựa cú pháp

Sản xuất	Quy tắc ngữ nghĩa
$L \rightarrow E \text{ return}$	Print (E.val)
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{Lexval}$

- Các ký hiệu E, T, F liên hệ với thuộc tính **tổng hợp** val
- Từ tố **digit** có thuộc tính tổng hợp lexval ( Được bộ phân tích từ vựng đưa ra )
- Định nghĩa tựa cú pháp này nhằm tính ra giá trị của biểu thức xuất phát từ giá trị lưu trữ trong bảng ký hiệu của các nhân tử là số

# Ví dụ DTC với thuộc tính kế thừa

Sản xuất	Quy tắc ngữ nghĩa
$D \rightarrow T\ L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L1.in = L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

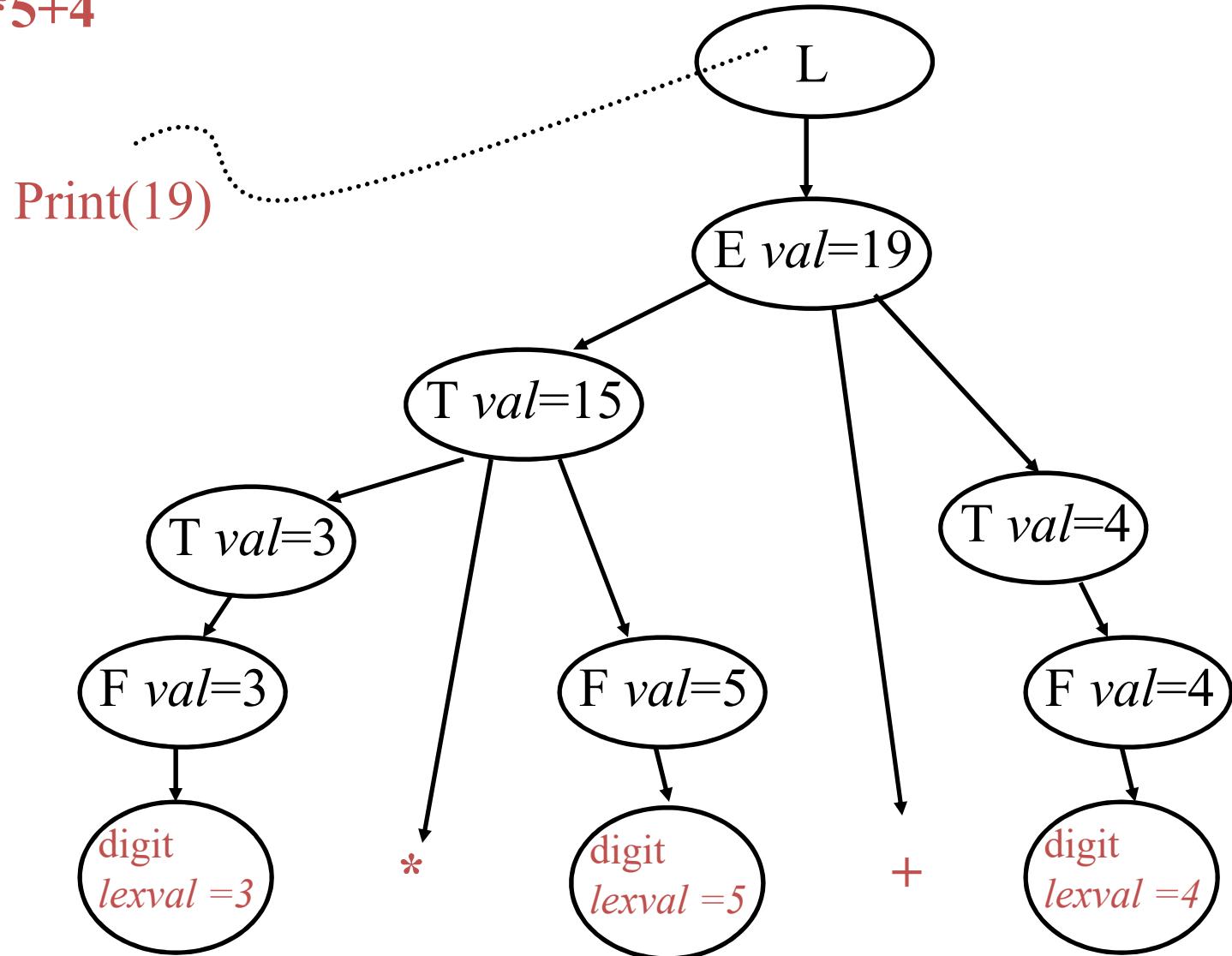
- $L.in$  là thuộc tính kế thừa, chỉ kiểu của danh sách biến  $L$
- Kiểu của danh sách biến  $L$  được tính từ kiểu của  $T$
- Kiểu trong thuộc tính  $L.in$  được truyền cho  $id$  cuối cùng và danh sách  $L1$  gồm các  $id$  còn lại Thông tin về kiểu được lưu vào bảng ký hiệu
- Thông tin kiểu được lưu cho tất cả các biến trong danh sách

## Cây phân tích cú pháp có chú giải

- Cây cú pháp có chỉ ra giá trị các thuộc tính tại mỗi nút được gọi là cây cú pháp có chú giải.

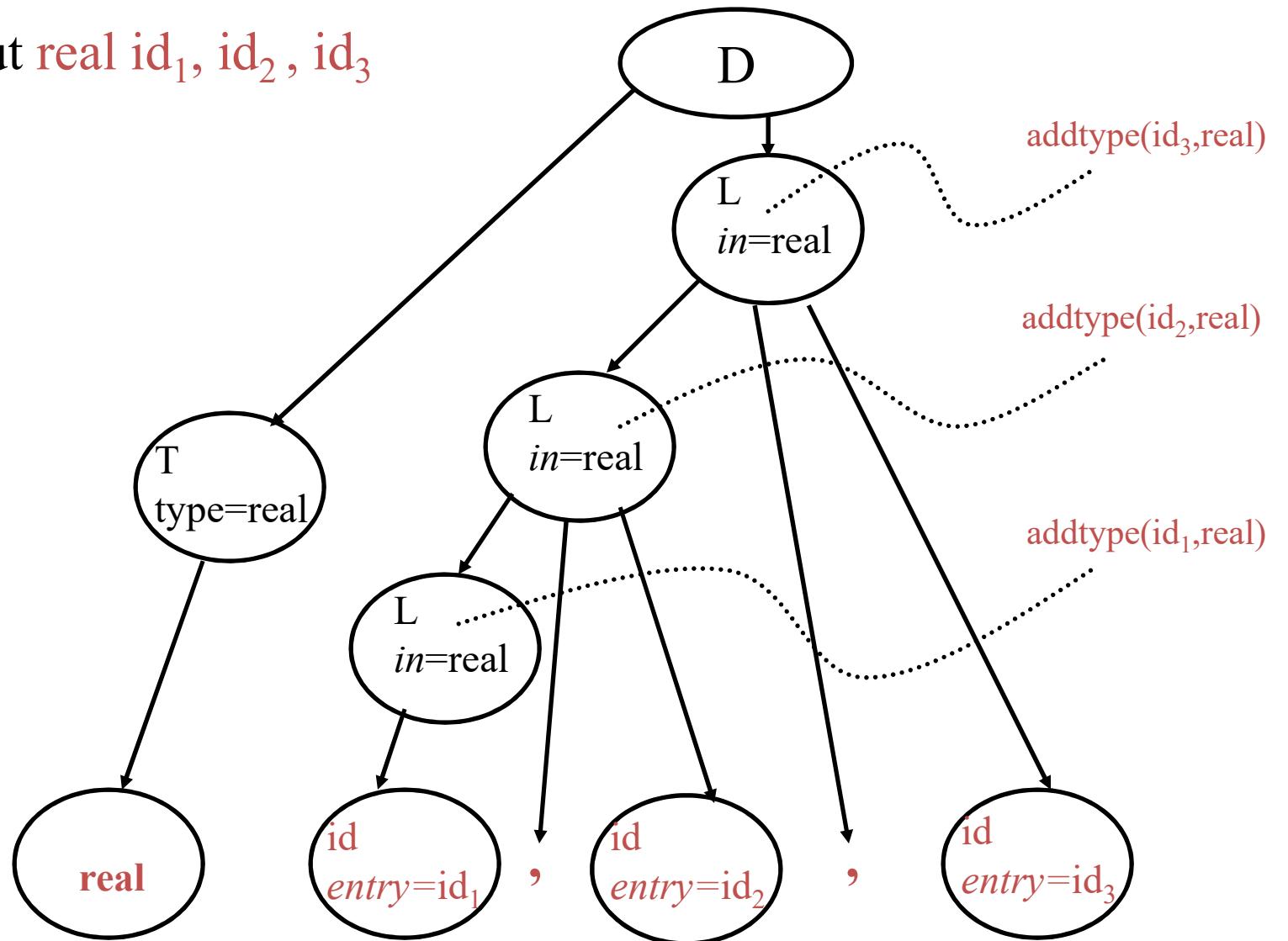
# Ví dụ: Cây PTCPCCG cho thuộc tính val

Input  $3*5+4$



## Ví dụ: Cây PTCPCCG cho ví dụ 2

Input  $\text{real } id_1, id_2, id_3$



# Đồ thị phụ thuộc

- Đồ thị định hướng
- Chỉ ra sự phụ thuộc lẫn nhau giữa các thuộc tính.
- Cách xây dựng:
  - Chuyển mọi quy tắc ngữ nghĩa về dạng  $b = f(c_1, \dots, c_k)$  bằng cách thêm vào thuộc tính tổng hợp dummy cho mọi quy tắc ngữ nghĩa là lời gọi thủ tục.
  - Chẳng hạn,
    - $L \rightarrow E \quad \text{print}(E.\text{val})$
    - Trở thành:  $\text{dummy} = \text{print}(E.\text{val})$
    - V.v....

# Xây dựng đồ thị phụ thuộc

*for each* nút  $n$  trong cây PTCP *do*

*for each* thuộc tính  $a$  của ký hiệu văn phạm  
tại  $n$  *do*

    tạo một nút trên đồ thị phụ thuộc cho  $a$

*for each* nút  $n$  trong cây phân tích cú pháp *do*

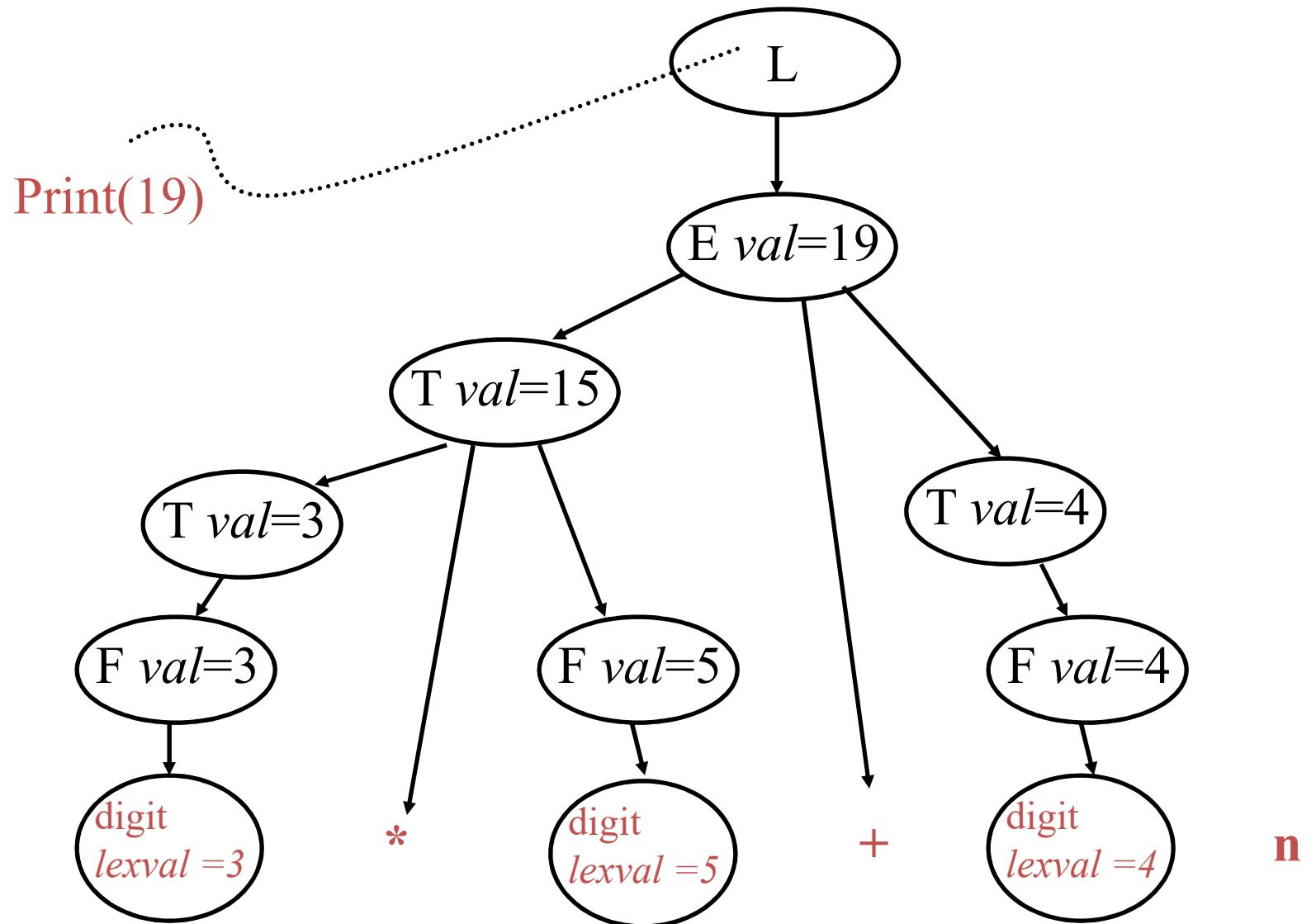
*for each* quy tắc ngữ nghĩa  $b = f(c_1, \dots, c_n)$

    liên hệ với sản xuất sử dụng tại nút  $n$  *do*

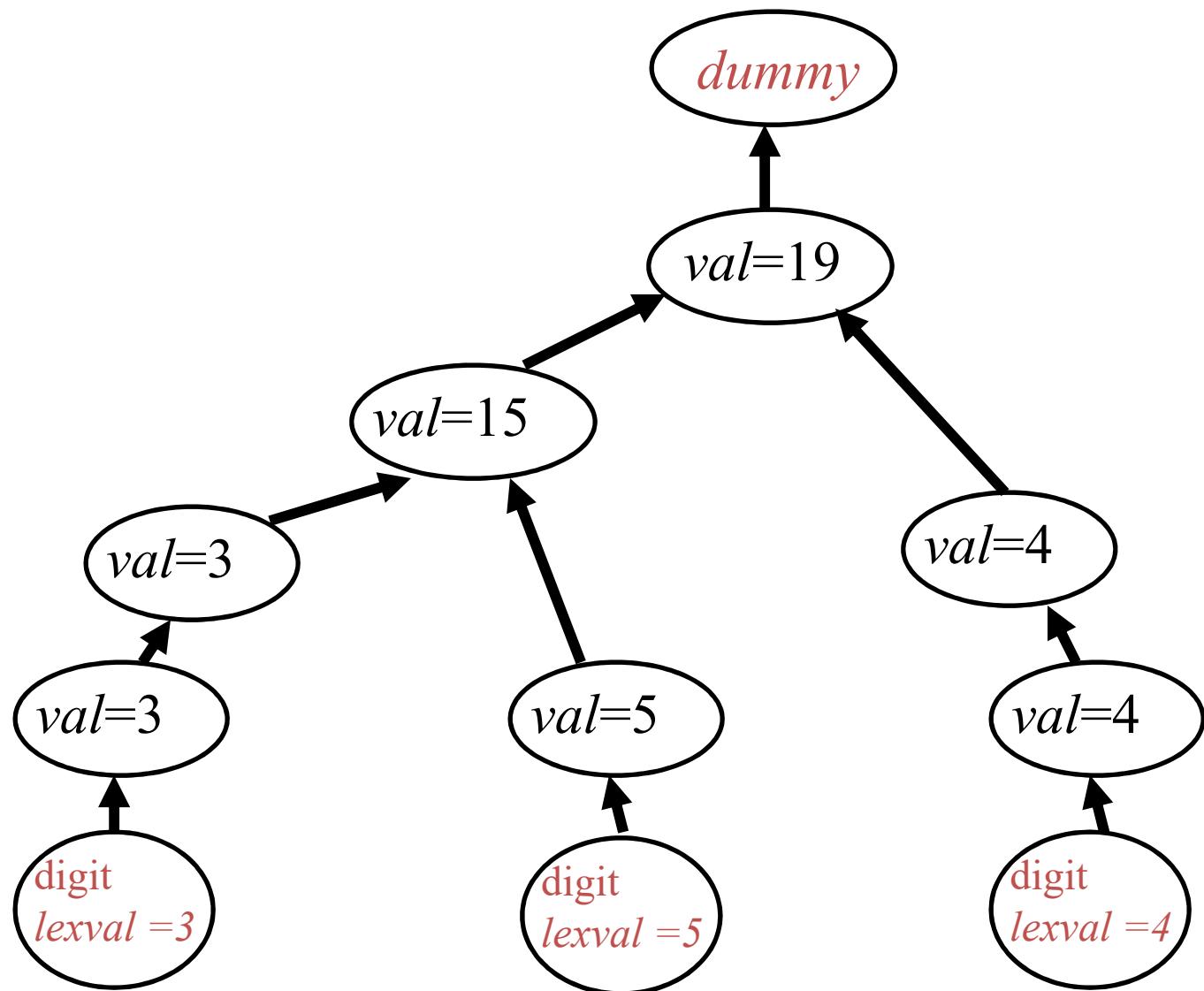
*for*  $i = 1$  *to*  $n$  *do*

            tạo một cung từ nút cho  $c_i$  tới  
            nút cho  $b$

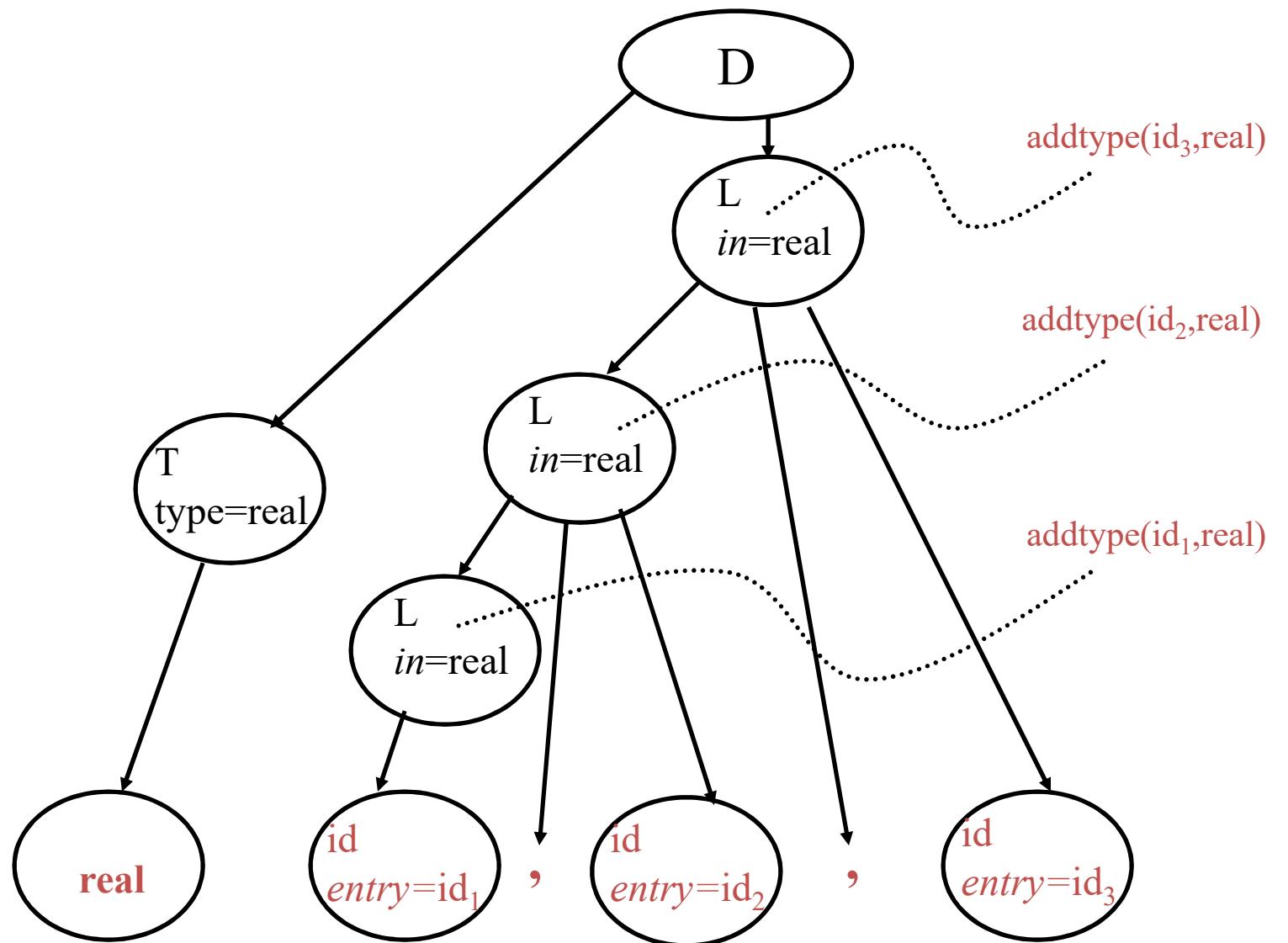
# Ví dụ 1



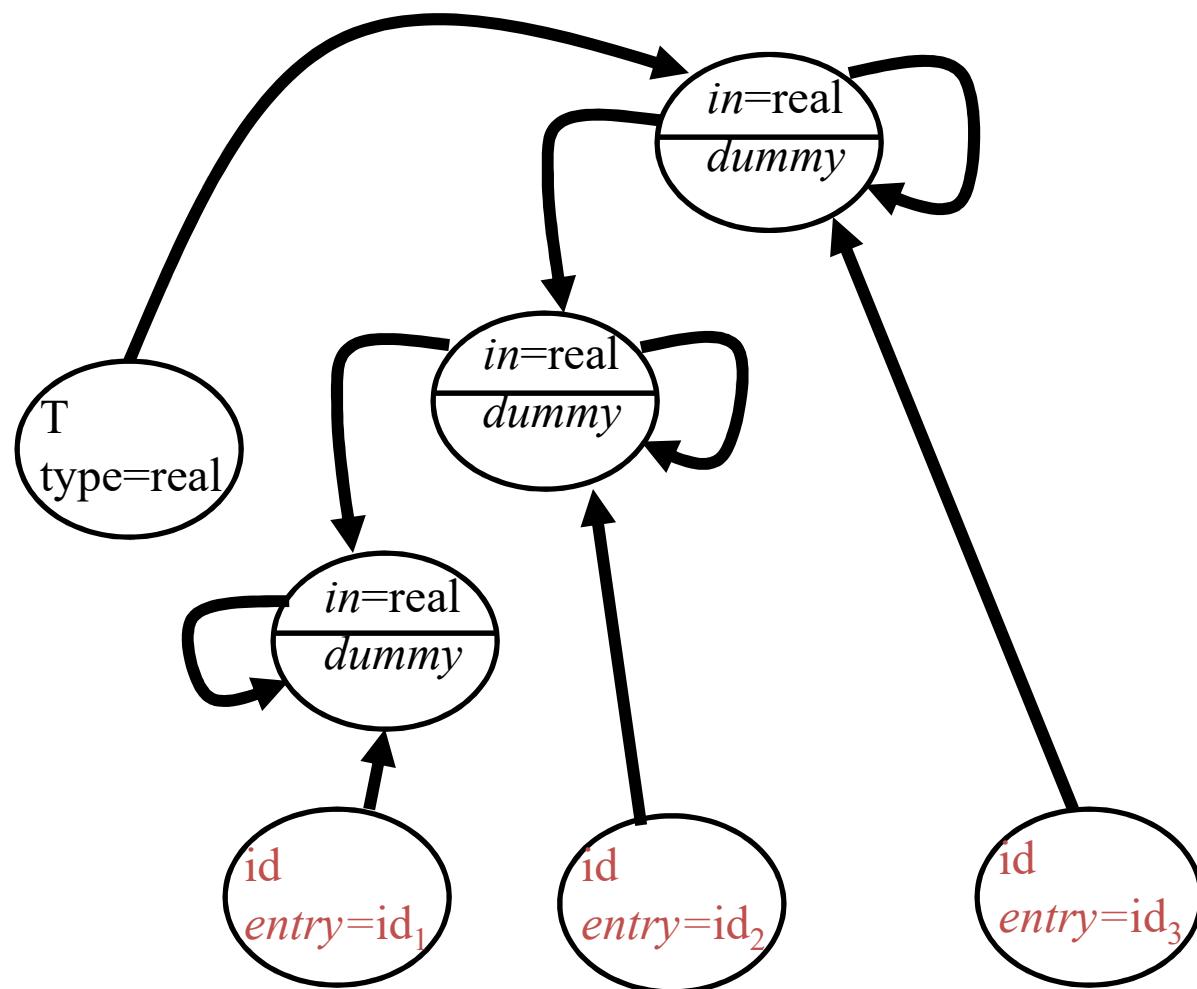
# Ví dụ 1



## Ví dụ 2



## Ví dụ 2



# Hệ thống kiểu

- Vấn đề kiểm tra kiểu
- Biểu thức và cấu trúc kiểu
- Hệ thống kiểu
- Chuyển đổi kiểu

# Vấn đề kiểm tra kiểu

- Kiểm tra xem chương trình có tuân theo các luật về kiểu của ngôn ngữ không
- Trình biên dịch quản lý thông tin về kiểu
- Việc kiểm tra kiểu được thực hiện bởi bộ kiểm tra kiểu (type checker), một bộ phận của trình biên dịch

## Ví dụ về kiểm tra kiểu

- Toán tử % của C chỉ thực hiện khi các toán hạng là số nguyên
- Chỉ có mảng mới có chỉ số và kiểu của chỉ số phải nguyên
- Một hàm phải có một số lượng tham số nhất định và các tham số phải đúng kiểu

# Kiểm tra kiểu

- Có hai phương pháp tĩnh và động
- Phương pháp áp dụng trong thời gian dịch là tĩnh
- Trong các ngôn ngữ như C hay Pascal, kiểm tra kiểu là tĩnh và được dùng để kiểm tra tính đúng đắn của chương trình trước khi nó được thực hiện
- Kiểm tra kiểu tĩnh cũng được sử dụng khi xác định dung lượng bộ nhớ cần thiết cho các biến
- Bộ kiểm tra kiểu được xây dựng dựa trên
  - Các biểu thức kiểu của ngôn ngữ
  - Bộ luật để định kiểu cho các cấu trúc

# Biểu thức kiểu (Type Expression)

Biểu diễn kiểu của một cấu trúc ngôn ngữ

Một biểu thức kiểu là một kiểu dữ liệu chuẩn hoặc được xây dựng từ các kiểu dữ liệu khác bởi cấu trúc kiểu (*Type Constructor*)

1. Kiểu dữ liệu chuẩn (int, real, boolean, char) là biểu thức kiểu
2. Biểu thức kiểu có thể liên hệ với một tên. Tên kiểu là biểu thức
3. Cấu trúc kiểu được ứng dụng vào các biểu thức kiểu tạo ra biểu thức kiểu

# Cấu trúc kiểu

(a) **Mảng (Array)**. Nếu  $T$  là biểu thức kiểu thì  $array(I, T)$  là biểu thức kiểu biểu diễn một mảng với các phần tử kiểu  $T$  và chỉ số trong miền  $I$

Ví dụ : `array [10] of integer` có kiểu `array(1..10,int);`

- (b) **Tích Descarter** Nếu  $T_1$  và  $T_2$  là các biểu thức kiểu thì tích Descarter  $T_1 \times T_2$  là biểu thức kiểu
- (c) **Bản ghi (Record)** Tương tự như tích Descarter nhưng chứa các tên khác nhau cho các kiểu khác nhau,

Ví dụ

```
struct
{
    double r;
    int i;
}
```

Có kiểu  $((r \times \text{double}) \times (i \times \text{int}))$

## Cấu trúc kiểu (tiếp)

- (d) *Con trả*: Nếu T là biểu thức kiểu thì  $\text{pointer}(T)$  là biểu thức kiểu
- (e) *Hàm Nếu D là miền xác định và R là miền giá trị của hàm thì kiểu của nó được biểu diễn là*  
*biểu\_thức : D : R.*

Ví dụ hàm của C

int f(char a, b)

Có kiểu: *char × char : int.*

# Hệ thống kiểu (Type System)

- Tập các luật để xây dựng các biểu thức kiểu trong những phần khác nhau của chương trình
- Được định nghĩa thông qua định nghĩa tựa cú pháp
- Bộ kiểm tra kiểu thực hiện một hệ thống kiểu
- Ngôn ngữ định kiểu mạnh: Chương trình dịch kiểm soát được hết các lỗi về kiểu

# Bộ kiểm tra kiểu của định danh

SẢN XUẤT	QUY TẮC NGỮ NGHĨA
$D \rightarrow \text{id} : T$	$\text{addtype}(\text{id}.entry, T.type)$
$T \rightarrow \text{char}$	$T.type := \text{char}$
$T \rightarrow \text{int}$	$T.type := \text{int}$
$T \rightarrow \uparrow T_1$	$T.type := \text{pointer}(T_1.type)$
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$	$T.type := \text{array}(1..\text{num}.val, T_1.type)$

# Bộ kiểm tra kiểu của biểu thức

SẢN XUẤT	QUY TẮC NGỮ NGHĨA
$E \rightarrow \text{literal}$	$E.type := \text{char}$
$E \rightarrow \text{num}$	$E.type := \text{int}$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \bmod E_2$	$E.type := \begin{aligned} &\text{if } E_1.type = \text{int} \text{ and } E_2.type = \text{int} \\ &\quad \text{then } \text{int} \\ &\quad \text{else } \text{type\_error} \end{aligned}$
$E \rightarrow E_1[E_2]$	$E.type := \begin{aligned} &\text{if } E_2.type = \text{int} \text{ and } E_1.type = \text{array}(s,t) \\ &\quad \text{then } t \\ &\quad \text{else } \text{type\_error} \end{aligned}$
$E \rightarrow E_1 \uparrow$	$E.type := \begin{aligned} &\text{if } E_1.type = \text{pointer}(t) \text{ then } t \\ &\quad \text{else } \text{type\_error} \end{aligned}$

# Bộ kiểm tra kiểu của lệnh

SẢN XUẤT	QUY TẮC NGỮ NGHĨA
$S \rightarrow \text{id} := E$	$S.type := \text{if } \text{id.type} = E.type \text{ then } void$ $\text{else } type\_error$
$S \rightarrow \text{if } E \text{ then } S_1$	$S.type := \text{if } E.type = boolean \text{ then } S_1.type$ $\text{else } type\_error$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.type := \text{if } E.type = boolean \text{ then } S_1.type$ $\text{else } type\_error$
$S \rightarrow S_1; S_2$	$S.type := \text{if } S_1.type = void \text{ and } S_2.type = void$ $\text{then } void$ $\text{else } type\_error$

# Bộ kiểm tra kiểu của hàm

SẢN XUẤT	QUY TẮC NGỮ NGHĨA
$D \rightarrow \text{id} : T$	$\text{addtype}(\text{id.entry}, T.type); D.type := T.type$
$D \rightarrow D_1; D_2$	$D.type := D_1.type \times D_2.type$
$\text{Fun} \rightarrow \text{fun id}(D) : T; B$	$\text{addtype}(\text{id.entry}, D.type : T.type)$
$B \rightarrow \{S\}$	
$S \rightarrow \text{id}(EList)$	$E.type := \begin{cases} \text{if } \text{lookup}(\text{id.entry}) = t_1 : t_2 \text{ and } EList.type = t_1 \\ \text{then } t_2 \\ \text{else } \text{type\_error} \end{cases}$
$EList \rightarrow E$	$EList.type := E.type$
$EList \rightarrow EList, E$	$EList.type := EList_1.type \times E.type$

# Hàm kiểm tra biểu thức kiểu tương đương

```
function sequiv(s, t): boolean;
begin
    if s và t là cùng kiểu dữ liệu chuẩn then
        return true;
    else if s = array(s1, s2) and t = array(t1, t2) then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = s1 x s2 and t = t1 x t2) then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1)
    else if s = s1→ s2 and t = t1→ t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else
        return false;
end;
```

# Chuyển đổi kiểu

- Kiểu của  $x+i$  với

x kiểu real

i kiểu int

Khi dịch sang lệnh máy, phép cộng với kiểu real và kiểu int có mã lệnh khác nhau

- Tùy ngôn ngữ và bộ luật chuyển đổi sẽ quy đổi các toán hạng về một trong hai kiểu

# Áp kiểu trong biểu thức

SẢN XUẤT	QUY TẮC NGỮ NGHĨA
$E \rightarrow \text{num}$	$E.type := \text{int}$
$E \rightarrow \text{num.num}$	$E.type := \text{real}$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ op } E_2$	$E.type := \begin{aligned} &\text{if } E_1.type = \text{int} \text{ and } E_2.type = \text{int} \\ &\quad \text{then } \text{int} \\ &\text{else if } E_1.type = \text{int} \text{ and } E_2.type = \text{real} \\ &\quad \text{then } \text{real} \\ &\text{else if } E_1.type = \text{real} \text{ and } E_2.type = \text{int} \\ &\quad \text{then } \text{real} \\ &\text{else if } E_1.type = \text{real} \text{ and } E_2.type = \text{real} \\ &\quad \text{then } \text{real} \\ &\text{else } \text{type\_error} \end{aligned}$

## 4.4. Sinh mã trung gian

- Mã ba địa chỉ
- Sinh mã cho lệnh gán
- Sinh mã cho các biểu thức logic
- Sinh mã cho các cấu trúc lập trình

# Mã trung gian

- Một chương trình với mã nguồn được chuyển sang chương trình tương đương trong ngôn ngữ trung gian bằng bộ sinh mã trung gian.
- Ngôn ngữ trung gian được người thiết kế trình biên dịch quyết định, có thể là:
  - Cây cú pháp
  - Ký pháp Ba Lan sau (hậu tố)
  - Mã 3 địa chỉ ...

# Mã trung gian

- Được sản sinh dưới dạng một chương trình cho một máy trùu tượng
- Mã trung gian thường dùng : mã ba địa chỉ, tương tự mã assembly
- Chương trình là một dãy các lệnh. Mỗi lệnh gồm tối đa 3 định danh.
- Tồn tại nhiều nhất một toán tử ở về phải cộng thêm một toán tử gán
  - Tên trung gian phải được sinh để thực hiện các phép toán trung gian
  - Các địa chỉ được thực hiện như con trỏ tới lối vào của nó trong bảng ký hiệu

## Mã trung gian của $t2=x + y * z$

- $t_1 := y * z$
- $t_2 := x + t_1$

# Các dạng mã ba địa chỉ phổ biến

- Mã 3 địa chỉ tương tự mã Assembly: lệnh có thể có nhãn, có những lệnh chuyển điều khiển cho các cấu trúc lập trình.
  1. *Lệnh gán*  $x := y \text{ op } z$ .
  2. *Lệnh gán* với phép toán 1 ngôi :  $x := op y$ .
  3. *Lệnh sao chép*:  $x := y$ .
  4. *Lệnh nhảy không điều kiện*:  $goto L$ ,  $L$  là nhãn của một lệnh
  5. *Lệnh nhảy có điều kiện*  $x \text{ relop } y \text{ goto } L$ .

## Các dạng mã ba địa chỉ

6. Lời gọi thủ tục param x và call p,n để gọi thủ tục p với n tham số . Return y là giá trị thủ tục trả về

param  $x_1$

param  $x_2$

...

param  $x_n$

Call p,n

7. Lệnh gán có chỉ số  $x:=y[i]$  hay  $x[i]:=y$

# Sinh mã trực tiếp từ ĐNTCP

- Thuộc tính tổng hợp S.code biểu diễn mã ba địa chỉ của lệnh
- Các tên trung gian được sinh ra cho các tính toán trung gian
- Các biểu thức được liên hệ với hai thuộc tính tổng hợp
  - E.place chứa địa chỉ chứa giá trị của E
  - E.code mã ba địa chỉ để đánh giá E
- Hàm newtemp sinh ra các tên trung giant1, t2,.. .
- Hàm gen sinh mã ba địa chỉ
- Trong thực tế, code được gửi vào file thay cho thuộc tính code

# Dịch trực tiếp cú pháp thành mã 3 địa chỉ

## Sản xuất

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow - E_1$

$E \rightarrow ( E_1 )$

$E \rightarrow id$

## Quy tắc ngữ nghĩa

{  $S.code = E.code \parallel gen(id.place := E.place)$  }

{  $E.place = newtemp$  ;

$E.code = E_1.code \parallel E_2.code \parallel$

$\parallel gen(E.place := E_1.place + E_2.place)$  }

{  $E.place = newtemp$  ;

$E.code = E_1.code \parallel E_2.code \parallel$

$\parallel gen(E.place := E_1.place * E_2.place)$  }

{  $E.place = newtemp$  ;

$E.code = E_1.code \parallel$

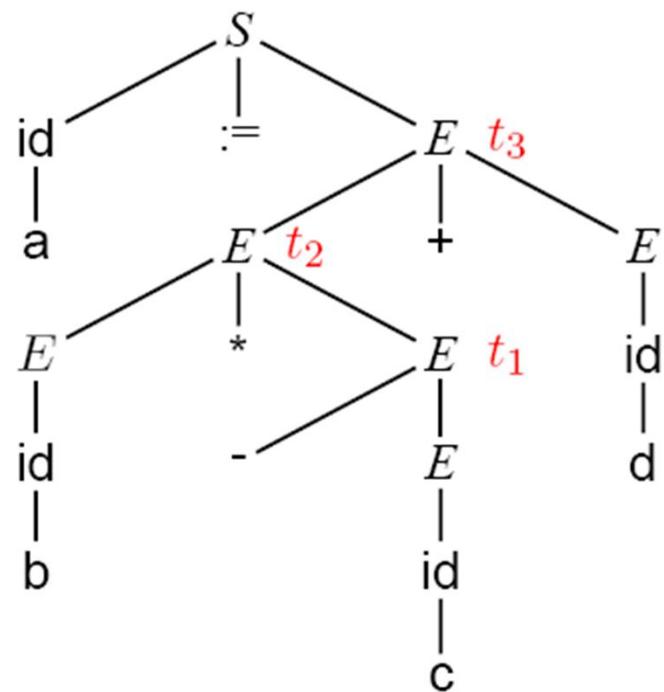
$\parallel gen(E.place := 'uminus' E_1.place)$  }

{  $E.place = E_1.place$  ;  $E.code = E_1.code$  }

{  $E.place = id.place$  ;  $E.code = "$  }

- Hàm ***newtemp*** trả về một dãy các tên khác nhau  $t_1, t_2 \dots$  cho lời gọi kế tiếp.
  - $E.place$ : là tên sẽ giữ giá trị của  $E$
  - $E.code$ : là dãy các câu lệnh 3 địa chỉ dùng để ước lượng  $E$

# Mã cho lệnh gán $a := b * -c + d$



$t_1 := \text{uminus } c$   
 $t_2 := b * t_1$   
 $t_3 := t_2 + d$   
 $a := t_3$

# Cài đặt câu lệnh 3 địa chỉ

## Bộ bốn (Quadruples)

$t_1 := -c$   
 $t_2 := b * t_1$   
 $t_3 := -c$   
 $t_4 := b * t_3$   
 $t_5 := t_2 + t_4$   
 $a := t_5$

	<i>op</i>	<i>arg1</i>	<i>arg2</i>	<i>result</i>
(0)	uminus	c		$t_1$
(1)	*	b	$t_1$	$t_2$
(2)	uminus	c		$t_3$
(3)	*	b	$t_3$	$t_4$
(4)	+	$t_2$	$t_4$	$t_5$
(5)	$:=$	$t_5$		a

Tên tạm phải được thêm vào bảng kí hiệu khi chúng  
được tạo ra.

# Cài đặt câu lệnh 3 địa chỉ

- Bộ ba (Triples)

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Tên tạm không được thêm vào trong bảng kí hiệu.

# Các dạng khác của câu lệnh 3 địa chỉ

- Ví dụ:

$x[i]:=y$   $x:=y[i]$

- Sử dụng 2 cấu trúc bộ ba

	$op$	$arg1$	$arg2$
(0)	[ ]	x	i
(1)	:=	(0)	y

	$op$	$arg1$	$arg2$
(0)	[ ]	y	i
(1)	:=	x	(0)

# Cài đặt câu lệnh 3 địa chỉ

- Bộ 3 gián tiếp: sử dụng một danh sách các con trỏ các bộ 3

	<i>op</i>		<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

# Sinh mã cho khai báo

Sử dụng biến toàn cục *offset*.

Các tên cục bộ trong chương trình con được truy xuất thông qua địa chỉ tương đối *offset*.

<u>Sản xuất</u>	<u>Quy tắc ngữ nghĩa</u>
$P \rightarrow M\ D$	{ }
$M \rightarrow \epsilon$	{ <i>offset</i> :=0 }
$D \rightarrow D; D$	
$D \rightarrow id : T$	{ <i>enter</i> ( <i>id.name</i> , <i>T.type</i> , <i>offset</i> ) <i>offset</i> := <i>offset</i> + <i>T.width</i> }
$T \rightarrow integer$	{ <i>T.type</i> = <i>integer</i> ; <i>T.width</i> = 4 }
$T \rightarrow real$	{ <i>T.type</i> = <i>real</i> ; <i>T.width</i> = 8 }
$T \rightarrow array [ num ] of T_1$	{ <i>T.type</i> = <i>array</i> (1.. <i>num.val</i> , <i>T<sub>1</sub>.type</i> ) <i>T.width</i> = <i>num.val</i> * <i>T<sub>1</sub>.width</i> }

# Lưu trữ thông tin về phạm vi

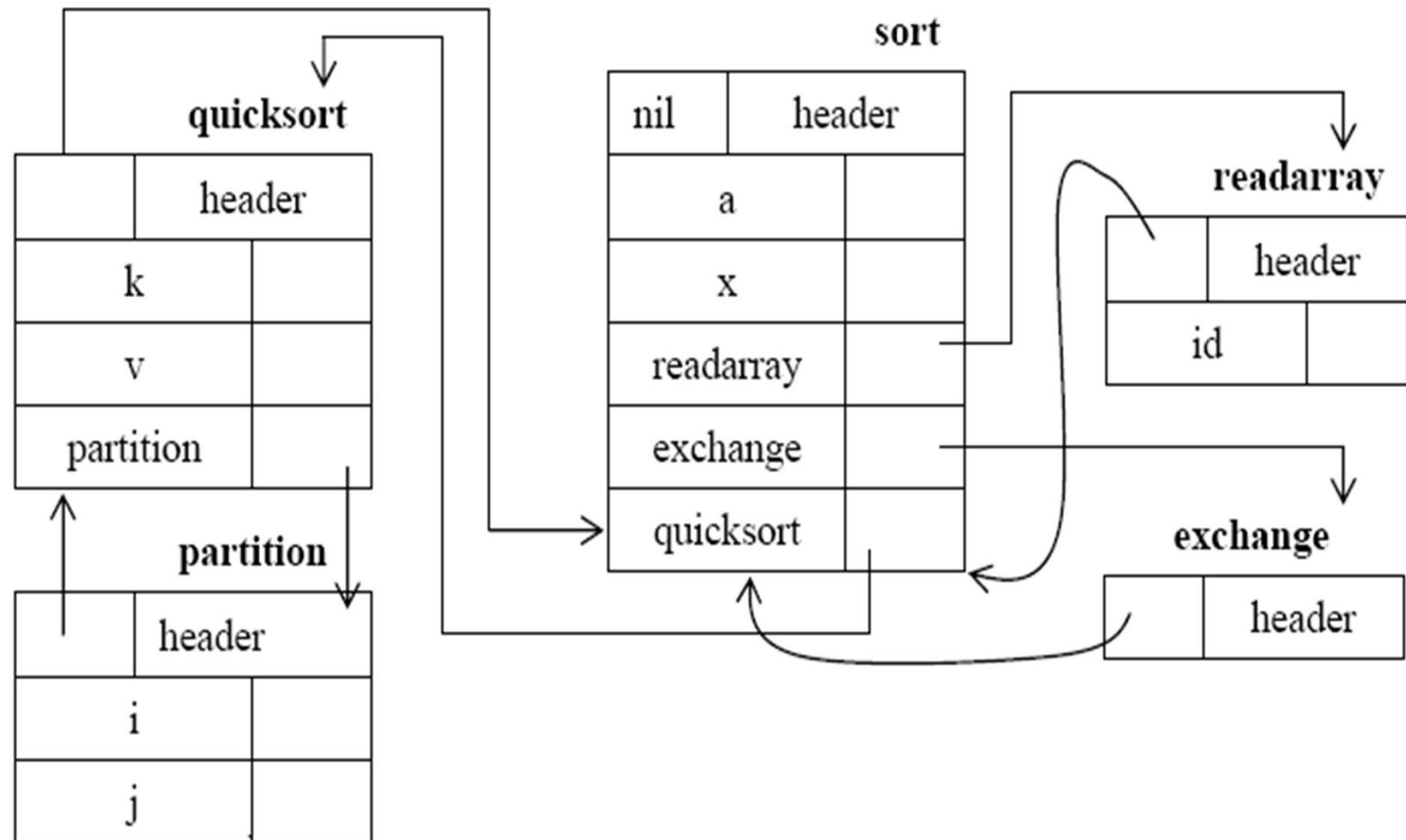
- Trong một ngôn ngữ mà chương trình con được phép khai báo lồng nhau, mỗi khi tìm thấy một CTC thì quá trình khai báo của chương trình bao nó bị tạm dừng.
- Văn phạm của khai báo này:  
$$P \rightarrow D$$
$$D \rightarrow D; D \mid id : T \mid proc\ id ; D ; S$$
- Khi một khai báo chương trình con  $D \rightarrow proc\ id\ D1; S$  được tạo ra thì các khai báo trong  $D1$  được lưu trong bảng kí hiệu mới.

# Khai báo chương trình con lồng nhau

- Ví dụ chương trình:

- 1) Program sort;
- 2) Var *a*: array[0..10] of integer;
- 3)       *x*: integer;
- 4)       Procedure readarray;
- 5)       Var *i*: integer;
- 6)       Begin ...*a*... end {readarray};
- 7)       Procedure exchange(*i*, *j*: integer);
- 8)       Begin {exchange} end;
- 9)       Procedure quicksort(*m*, *n*: integer);
- 10)       Var *k*, *v*: integer;
- 11)       Function partition(*y*,*z*: integer): integer;
- 12)       Begin ...*a*...*v*..exchange(*i*,*j*) end; {partition}
- 13)       Begin ... end; {quicksort}
- 14) Begin ... end; {sort}

# Năm bảng kí hiệu của Sort



# Các thủ tục trong tập quy tắc ngũ nghĩaS

**mktable(previous)** – tạo một bảng kí hiệu mới, bảng này có previous chỉ đến bảng cha của bảng kí hiệu mới này.

**enter(table,name,type,offset)** – tạo ra một ô mới có tên *name* trong bảng kí hiệu được chỉ ra bởi *table* và đặt kiểu *type*, địa chỉ tương đối *offset* vào các trường bên trong ô đó.

**enterproc(table,name,newbtable)** – tạo ra một ô mới cho tên chương trình con vào *table*, *newbtable* trả tới bảng kí hiệu của chương trình con này.

**addwidth(table,width)** – ghi tổng kích thước của tất cả các ô trong bảng kí hiệu vào header của bảng đó.

## Xử lý các khai báo trong những chương trình con lồng nhau

$P \rightarrow M D \quad \{ addwidth(top(tblptr), top(offset)); pop(tblptr);$   
 $pop(offset) \}$

$M \rightarrow \epsilon \quad \{ t := mkttable(null); push(t, tblptr); push(0, offset) \}$

$D \rightarrow D_1 ; D_2$

$D \rightarrow \text{proc id ; N D}_1 ; S \quad \{ t := top(tblptr); addwidth(t, top(offset));$   
 $pop(tblptr); pop(offset);$   
 $enterproc(top(tblptr), id.name, t) \}$

$N \rightarrow \epsilon \quad \{ t := mkttable(top(tblptr)); push(t, tblptr); push(0, offset); \}$

$D \rightarrow \text{id : T} \{ enter(top(tblptr), id.name, T.type, top(offset));$   
 $top(offset) := top(offset) + T.width$

- **tblptr** – để giữ con trỏ bảng kí hiệu.
- **offset** – lưu trữ địa chỉ offset hiện tại của bảng kí hiệu trong **tblptr**.

# Xử lý các khai báo trong những chương trình con lồng nhau

- Với sản xuất  $A \rightarrow BC \{action_A\}$  thì các hoạt động trong cây con B, C được thực hiện trước A.
- Sản xuất  $M \rightarrow \epsilon$  khởi tạo stack *tblptr* với một bảng kí hiệu cho phạm vi ngoài cùng (chương trình sort) bằng lệnh ***mkttable(nil)*** đồng thời đặt offset = 0.
- N đóng vai trò tương tự M khi một khai báo chương trình con xuất hiện, nó dùng lệnh ***mkttable(top(tblptr))*** để tạo ra một bảng mới, tham số ***top(tblptr)*** cho giá trị con trả tới bảng lại được đẩy vào đỉnh stack *tblptr* và 0 được đẩy vào stack offset.
- Với mỗi khai báo ***id: T*** một ô mới được tạo ra cho *id* trong bảng kí hiệu hiện hành, stack *tblptr* không đổi, giá trị *top(offset)* được tăng lên bởi *T.width*.
- Khi ***D → proc id ; N D<sub>1</sub> ; S*** diễn ra thì kích thước của tất cả các đối tượng dữ liệu khai báo trong *D<sub>1</sub>* sẽ nằm trên đỉnh stack offset. Nó được lưu trữ bằng cách dùng Addwidth, các stack *tblptr* và offset bị đẩy và chúng ta thao tác trên các khai báo của chương trình con.

# Tên trong bảng kí hiệu

- Xét ĐNTCP để sinh ra mã lệnh 3 địa chỉ cho lệnh gán

$S \rightarrow id := E \quad \{ p := lookup(id.name);$   
 $if p \neq nil then emit(p := E.place) \ else error \}$

$E \rightarrow E_1 + E_2 \quad \{ E.place := newtemp;$   
 $emit(E.place := E_1.place + E_2.place) \}$

$E \rightarrow E_1 * E_2 \quad \{ E.place := newtemp;$   
 $emit(E.place := E_1.place * E_2.place) \}$

$E \rightarrow - E_1 \quad \{ E.place := newtemp;$   
 $emit(E.place := 'unimus' E_1.place) \}$

$E \rightarrow ( E_1 ) \quad \{ E.place := E_1.place \}$

$E \rightarrow id \quad \{ p := lookup(id.name);$   
 $if p \neq nil then E.place := p \ else error \}$

# Tên trong bảng kí hiệu

- Hàm lookup sẽ tìm trong bảng kí hiệu xem có hay không một tên được cho bởi *id.name*. Nếu có thì trả về con trỏ của ô, nếu không thì trả về nil.
- Thủ tục emit để đưa mã 3 địa chỉ vào một tập tin output chứ không xây dựng thuộc tính code cho các kí hiệu chưa kết thúc như gen. Quá trình dịch thực hiện bằng cách đưa ra một tập tin output nếu thuộc tính code của kí hiệu không kết thúc trong về trái sản xuất được tạo ra bằng cách nối thuộc tính code của kí hiệu không kết thúc trong về phải theo đúng thứ tự xuất hiện của các kí hiệu chưa kết thúc ở về phải.

# Tên trong bảng kí hiệu

- Xét sản xuất D → proc id; ND<sub>1</sub>; S
- Các tên trong lệnh gán sinh ra bởi kí hiệu không kết thúc S sẽ được khai báo trong chương trình con này hoặc trong chương trình chứa nó.
- Khi khai báo tới một tên thì trước hết hàm lookup sẽ tìm xem tên đó có trong bảng kí hiệu hiện hành hay không, nếu không thì dùng con trỏ trong header của bảng để tìm bảng kí hiệu bao nó và tìm trong đó, nếu không tìm thấy trong tất cả các mức thì lookup trả về nil.

# Địa chỉ hóa các phần tử của mảng

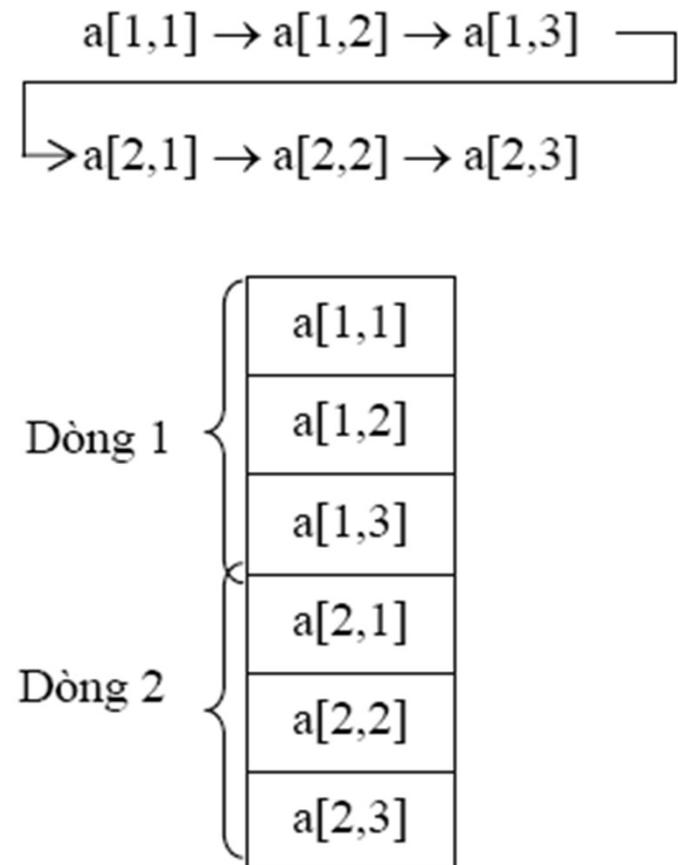
- Các phần tử của mảng có thể truy xuất nhanh nếu chúng được lưu trữ trong một khối ô nhớ kế tiếp nhau. Trong mảng một chiều, nếu kích thước của một phần tử là  $w$  thì địa chỉ tương đối phần tử thứ  $i$  của mảng  $A$  được tính theo công thức:
- $A[i] = \text{base} + (i-\text{low})*w$
- Trong đó:
  - Low: cận dưới tập chỉ số
  - Base: địa chỉ tương đối của ô nhớ cấp phát cho mảng (địa chỉ tương đối của  $A[\text{low}]$ )
- Tương đương  $A[i] = i*w + (\text{base} - \text{low})*w$
- Trong đó:
  - $c = \text{base} - \text{low} * w$  có thể được tính tại thời gian dịch và lưu trong bảng kí hiệu
- $\Rightarrow A[i] = i*w + c$

# Địa chỉ hóa các phần tử của mảng 2 chiều

- Theo dòng

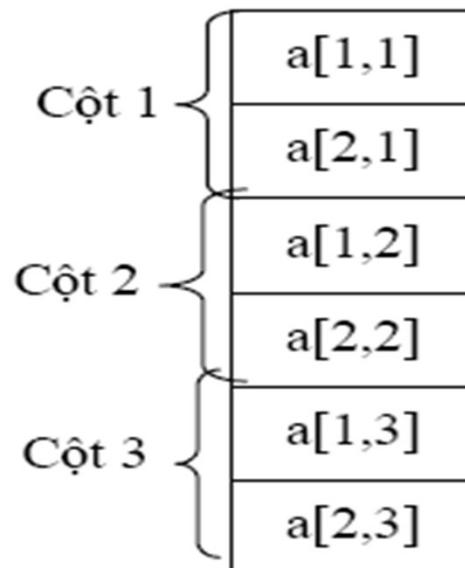
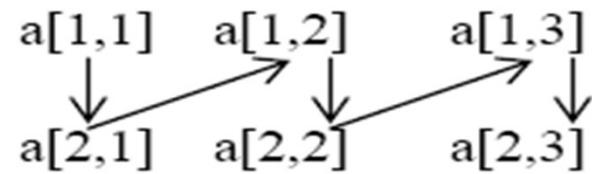
Địa chỉ tương đối của  $A[i_1, i_2] =$   
 $base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

$low_1, low_2$ : cận dưới cho  $i_1$  và  $i_2$   
 $n_2$ : số lượng các giá trị mà  $i_2$  có thể nhận. Nếu  $high_2$  là cận trên của  $i_2$  thì  $n_2 = high_2 - low_2 + 1$



# Địa chỉ hóa các phần tử của mảng 2 chiều

## – Theo cột



# Sinh mã biểu thức Logic

- Biểu thức logic được sinh bởi văn phạm sau:  
$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id} \mid \text{relop} \mid \text{id} \mid \text{true} \mid \text{false}$$
- Trong đó:
  - Or và And kết hợp trái
  - Or có độ ưu tiên thấp nhất tiếp theo là And, và Not

# Biểu diễn bằng số

- Mã hóa true và false bằng các số và ước lượng một biểu thức boole tương tự như đổi với biểu thức số học
  - Có thể biểu diễn true là 1; false là 0
  - Hoặc các số khác 0 là true, 0 là false
- Ví dụ: biểu thức a or b and not c
- Mã 3 địa chỉ:
    - t1 = not c
    - t2 = b and t1
    - t3 = a or t2
  - Biểu thức quan hệ a<b tương đương lệnh điều kiện if a<b then 1 else 0. Mã 3 địa chỉ tương ứng:
    - 100: if a<b goto 103
    - 101: t:=0
    - 102: goto 104
    - 103: t:= 1
    - 104:

# ĐNTCPdùng số để biểu diễn các giá trị logic

$E \rightarrow E_1 \text{ or } E_2 \quad \{ E.place := newtemp; \text{ emit}(E.place := 'E_1.place \text{ or } E_2.place) \}$

$E \rightarrow E_1 \text{ and } E_2 \quad \{ E.place := newtemp; \text{ emit}(E.place := 'E_1.place \text{ and } E_2.place) \}$

$E \rightarrow \text{not } E_1 \quad \{ E.place := newtemp; \text{ emit}(E.place := 'not' E_1.place) \}$

$E \rightarrow id_1 \text{ relop } id_2 \quad \{ E.place := newtemp; \text{ emit}(E.place := 'id_1.place \text{ relop } id_2.place) \}$

Nextstat cho biết  
nhân của câu  
lệnh 3 địa chỉ  
tiếp theo.

Emit: đặt câu lệnh  
3 địa chỉ vào tập  
tin, emit làm tăng  
nextstat sau khi  
thực hiện

$(\text{if } id_1.place \text{ relop.op } id_2.place \text{ goto' nextstat +3);}$   
 $(E.place := '0'); \text{ emit('goto' nextstat +2);}$   
 $(E.place := '1') \}$

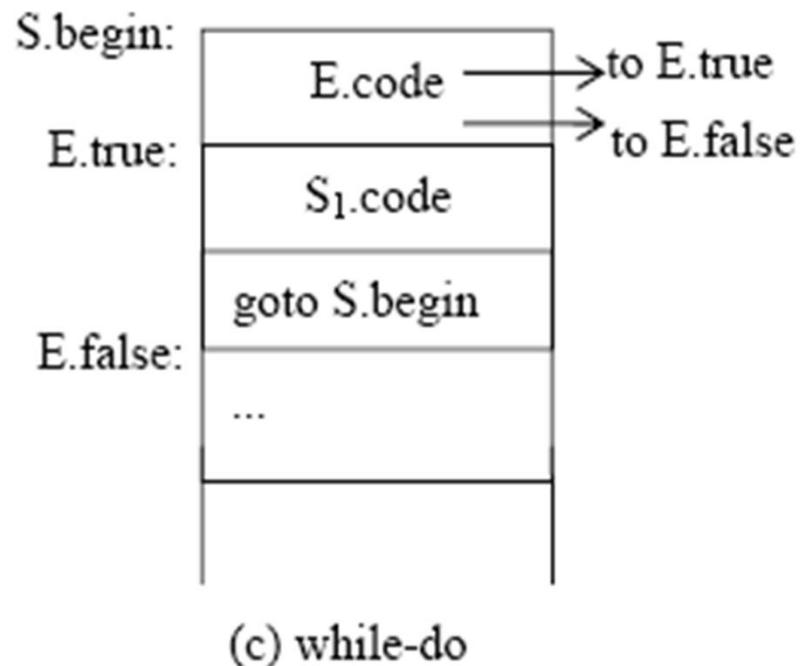
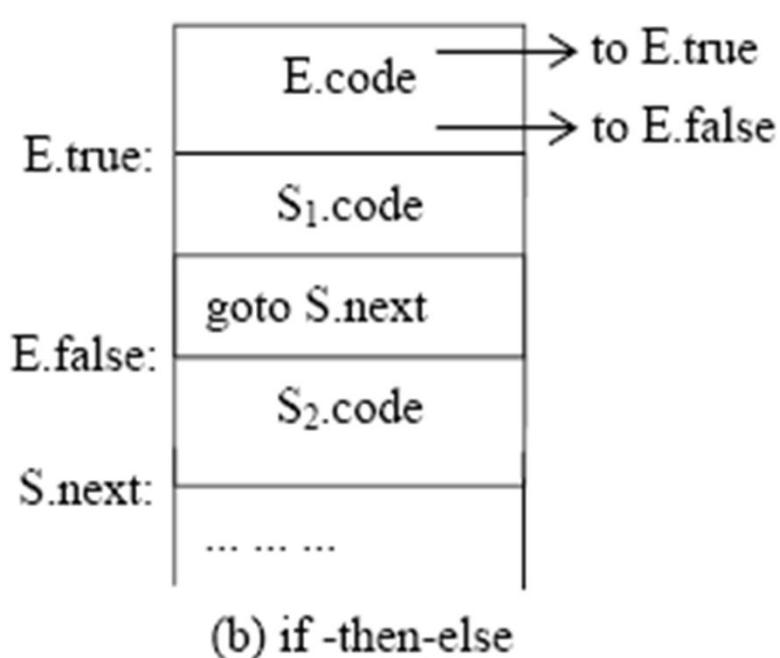
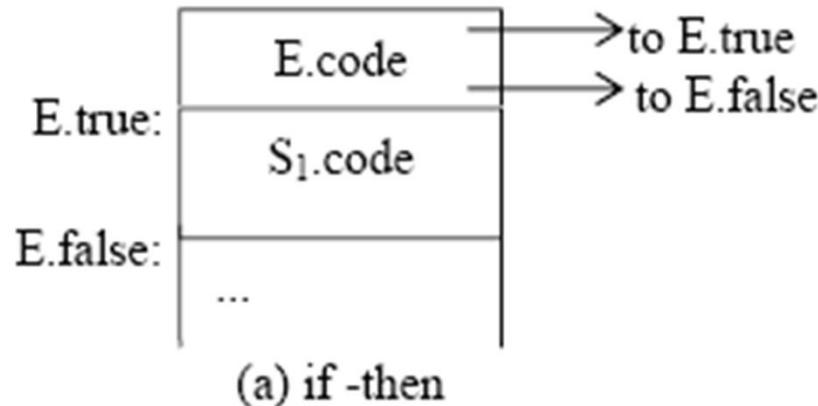
$E \rightarrow \text{true} \quad \{ E.place := newtemp; \text{ emit}(E.place := '1') \}$

$E \rightarrow \text{false} \quad \{ E.place := newtemp; \text{ emit}(E.place := '0') \}$

# Sinh mã cho các cấu trúc lập trình

- Biểu diễn các giá trị của biểu thức Boole bằng biểu thức đã đến được trong một chương trình.
- Ví dụ: cho câu lệnh sau
- $S \rightarrow \text{if } E \text{ then } S_1 \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S_1$
- Với mỗi biểu thức E chúng ta kết hợp với 2 nhãn:
  - E.true: nhãn của dòng điều khiển nếu E là true
  - E.false: nhãn của dòng điều khiển nếu E là false
  - S.code: mã lệnh 3 địa chỉ được sinh ra bởi S
  - S.next: là nhãn mã lệnh 3 địa chỉ đầu tiên sẽ thực hiện sau mã lệnh của S
  - S.begin: nhãn địa chỉ lệnh đầu tiên được sinh ra cho S

# Mã lệnh của các lệnh if-then, if-then-else, while-do



# ĐNTCP cho các cấu trúc lập trình

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if } B \text{ then } S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \parallel label(B.true) \parallel S_1.code$ $\quad \parallel \text{gen('goto' } S.next)$ $\quad \parallel label(B.false) \parallel S_2.code$
$S \rightarrow \text{while } B \text{ do } S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\quad \parallel label(B.true) \parallel S_1.code$ $\quad \parallel \text{gen('goto' } begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

## Dịch biểu thức logic trong các cấu trúc lập trình

- Nếu E có dạng:  $a < b$  thì mã lệnh sinh ra có dạng  
If  $a < b$  then goto E.true else goto E.false
- Nếu E có dạng:  $E_1 \text{ or } E_2$  thì
  - Nếu  $E_1$  là true thì E cũng là true
  - Nếu  $E_1$  là false thì phải đánh giá  $E_2$ ; E sẽ là true hay false phụ thuộc  $E_2$
- Tương tự với  $E_1 \text{ and } E_2$

# Dịch biểu thức logic trong các cấu trúc lập trình

SẢN XUẤT	QUY TẮC NGỮ NGHĨA
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code    \text{gen}(E.false ':')    E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code    \text{gen}(E.true ':')    E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow id_1 \text{ relop } id_2$	$E.code := \text{gen}('if' id_1.place relop.op id_2.place 'goto' E.true)    \text{gen}('goto' E.false)$
$E \rightarrow \text{true}$	$E.code := \text{gen}('goto' E.true)$
$E \rightarrow \text{false}$	$E.code := \text{gen}('goto' E.false)$

# Biểu thức logic ở dạng hỗn hợp

- Thực tế, các biểu thức logic thường chứa các biểu thức số học như trong  $(a+b) < c$

Xét văn phạm  $E \rightarrow E + E \mid E \text{ and } E \mid E \text{ relop } E \mid id$

Trong đó,  $E$  and  $E$  đòi hỏi hai đối số phải là logic. Trong khi  $+$  và relop có các đối số là biểu thức logic hoặc/số học.

Để sinh mã lệnh trong trường hợp này, chúng ta dùng thuộc tính tổng hợp  $E.type$  có thể là arith hoặc bool.  $E$  sẽ có các thuộc tính kế thừa  $E.true$  và  $E.false$  đối với biểu thức số học.

# Biểu thức logic ở dạng hỗn hợp

QT ngữ nghĩa kết hợp với  $E \rightarrow E_1 + E_2$

```
E.type := arith;  
if E1.type = arith and E2.type = arith then begin  
/* phép cộng số học bình thường */  
E.place := newtemp;  
E.code := E1.code || E2.code || gen(E.place ':=' E1.place +' E2.place)  
end  
else if E1.type = arith and E2.type = bool then begin  
E.place := newtemp;  
E2.true := newlabel;  
E2.false := newlabel;  
E.code := E1.code || E2.code || gen(E2.true ':=' E.place := ' E1.place +1) ||  
gen('goto' nextstat +1) || gen(E2.false ':=' E.place := ' E1.place)  
else if ...
```

# Biểu thức logic ở dạng hỗn hợp

nếu có biểu thức logic nào có biểu thức số học, sinh mã lệnh cho E1, E2 bởi

$E_2.\text{true} : E.\text{place} := E_1.\text{place} + 1$

$\text{goto nextstat} + 1$

$E_2.\text{false} : E.\text{place} := E_1.\text{place}$

## Ví dụ

- Sinh mã trung gian cho lệnh sau:

while a < b do

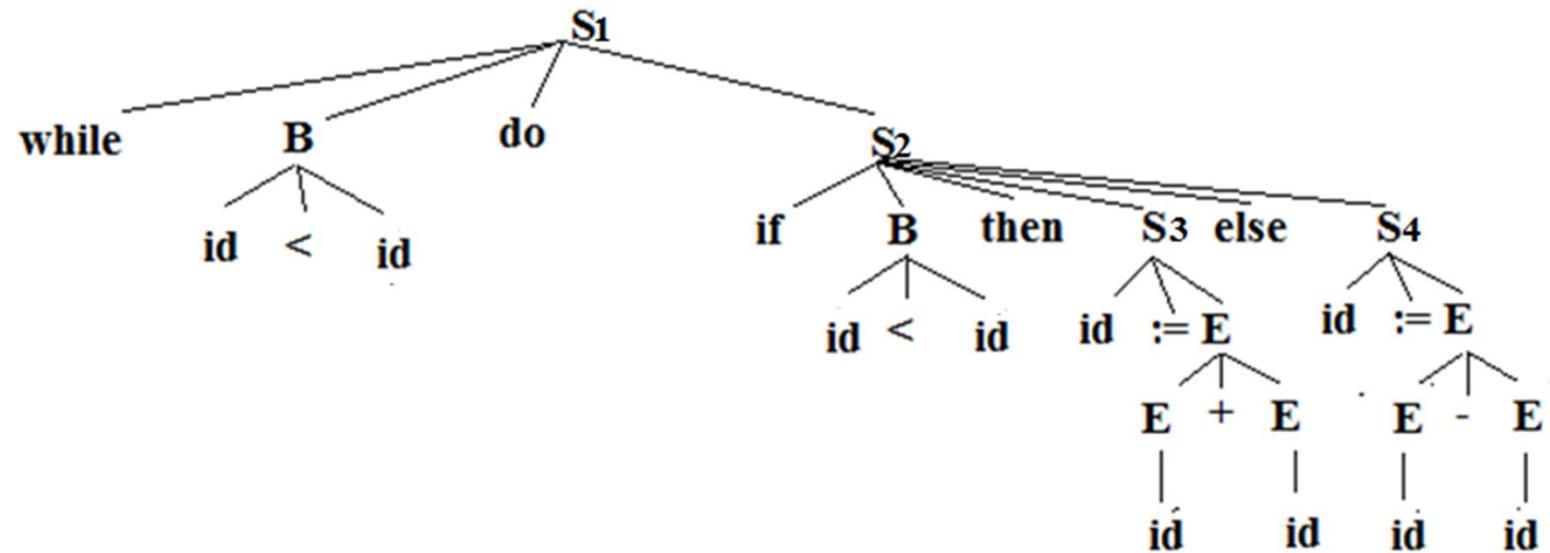
if c < d then

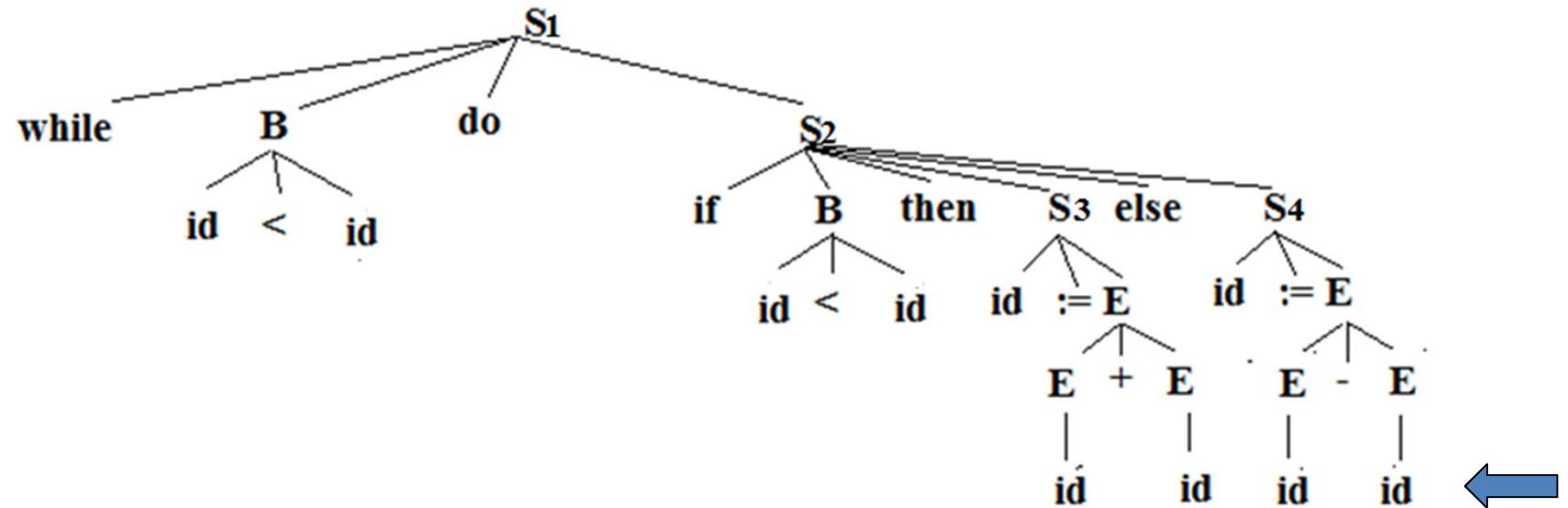
x := y + z

else

x := y - z

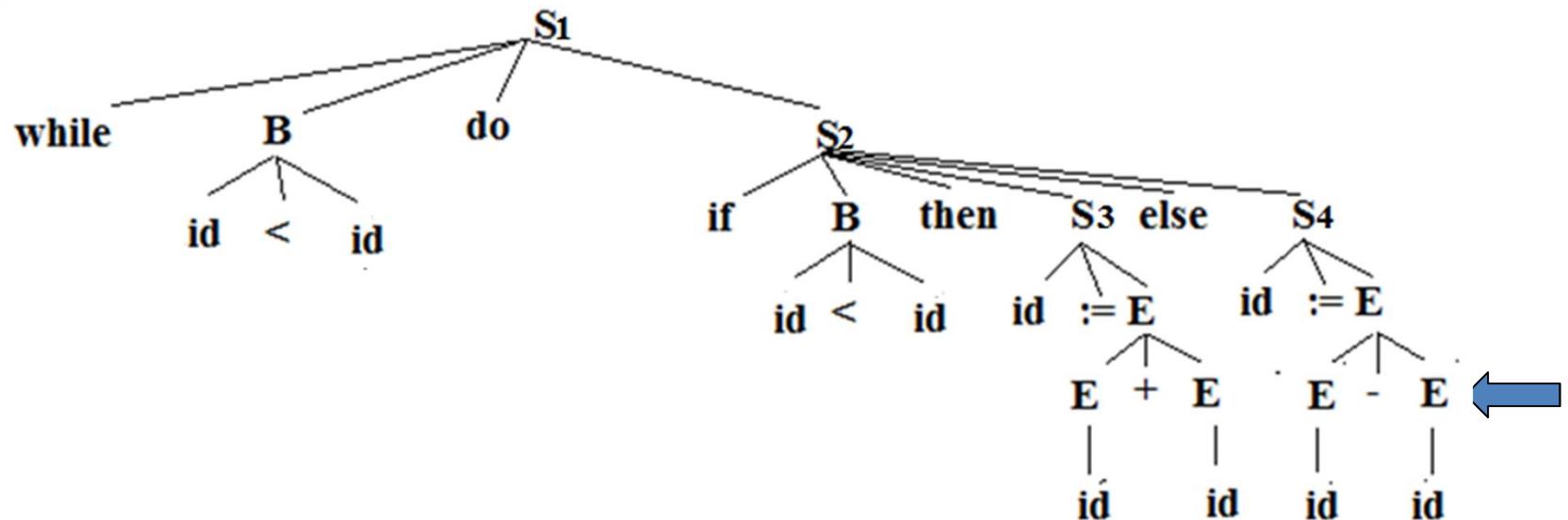
# Cây cú pháp của lệnh





Thuộc tính

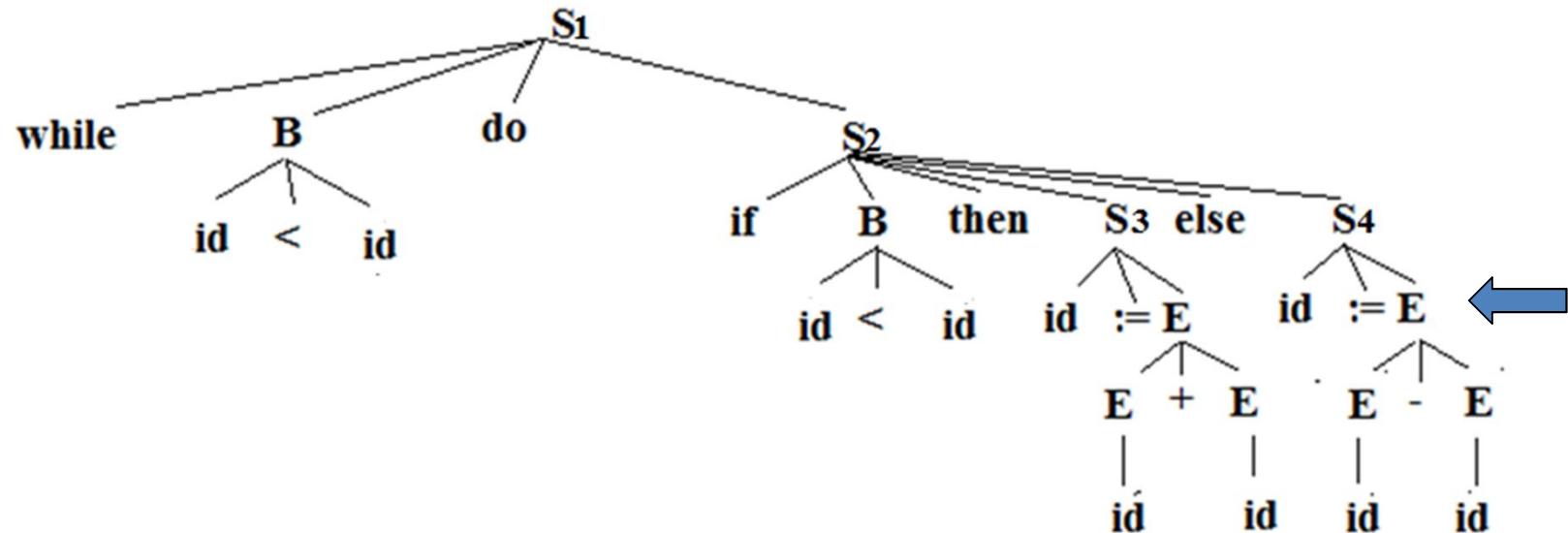
id.place = z



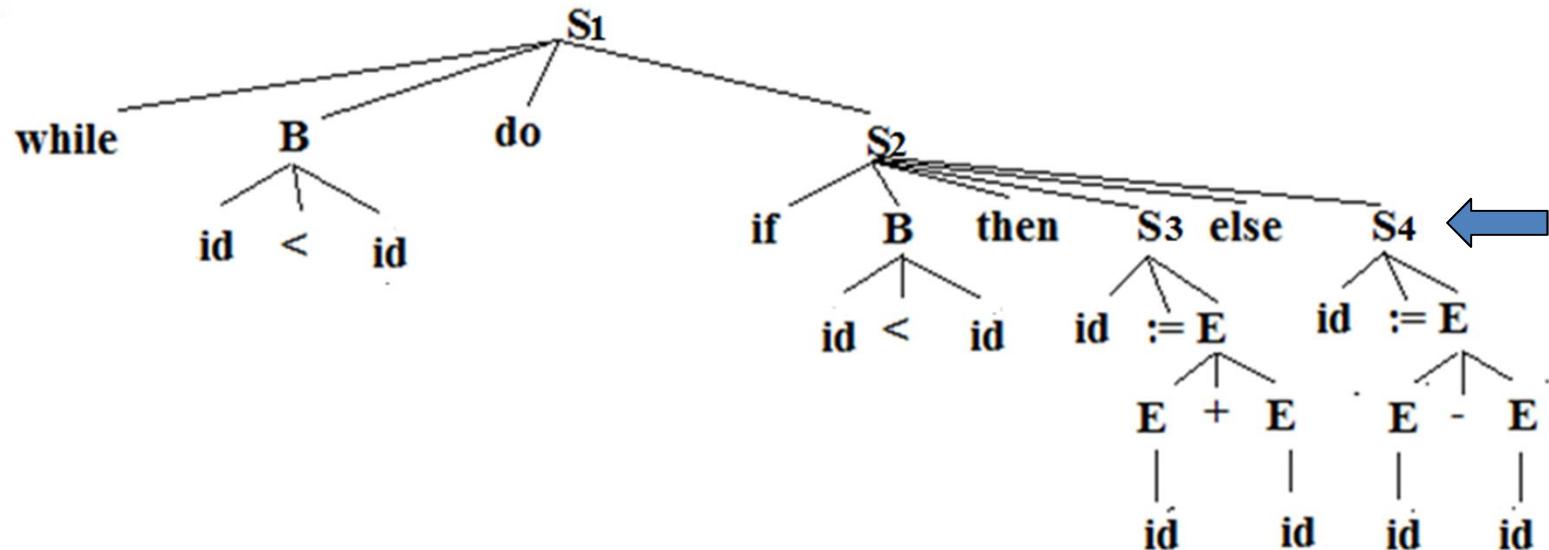
Thuộc tính

E.place = z

E.code = ""



**Thuộc tính**  
 $E.place = \text{newtemp}()$  t1  
 $E.code = "t1=y-z"$



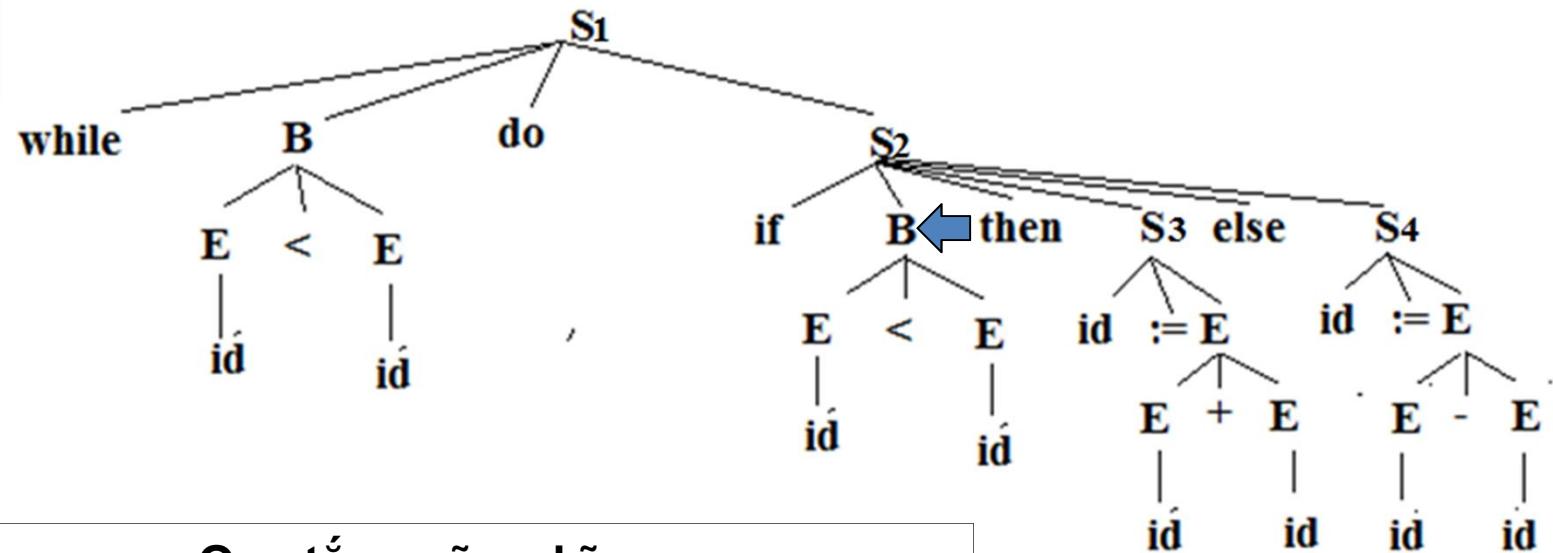
$S \rightarrow \text{id} := E$

Quy tắc ngữ nghĩa  
 $\{ S.\text{code} = E.\text{code} \parallel \text{gen}(\text{id.place} := E.\text{place}) \}$

### Thuộc tính

$\text{id.place} = x$

$S.\text{code} = "t1=y-z  
x=t1"$



### Quy tắc ngữ nghĩa

$E \rightarrow id_1 \text{ relop } id_2$

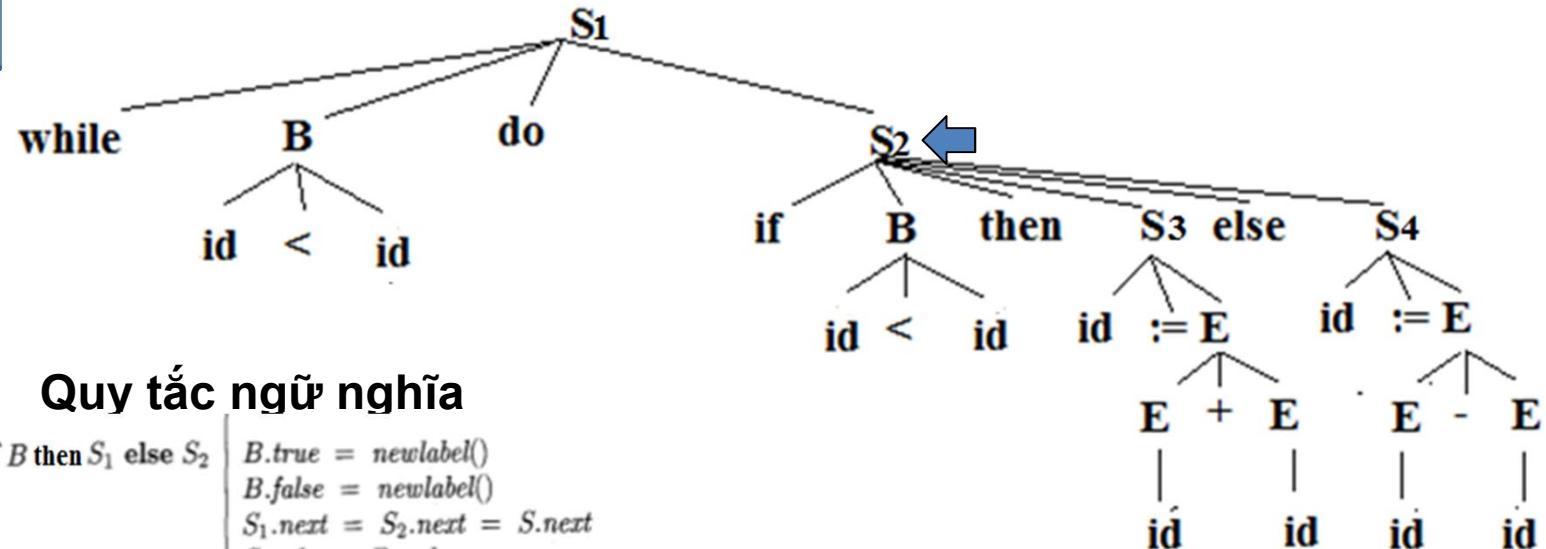
$E.\text{code} := \text{gen('if' } id_1.\text{place relop.op } id_2.\text{place } 'goto' E.\text{true}) \parallel \text{gen('goto' } E.\text{false})$

### Thuộc tính

.code

if c < d goto L1

Goto L2



### Quy tắc ngữ nghĩa

$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$      
  $B.\text{true} = \text{newlabel}()$   
 $B.\text{false} = \text{newlabel}()$   
 $S_1.\text{next} = S_2.\text{next} = S.\text{next}$   
 $S.\text{code} = B.\text{code}$   
 $\quad \quad \quad || \text{label}(B.\text{true}) || S_1.\text{code}$   
 $\quad \quad \quad || \text{gen('goto' } S.\text{next})$   
 $\quad \quad \quad || \text{label}(B.\text{false}) || S_2.\text{code}$

### Thuộc tính

$B.\text{true} = \text{newlabel}()$   $\rightarrow L1$

$B.\text{false} = \text{newlabel}()$   $\rightarrow L2$

$S1.\text{next} = S2.\text{next} = S3.\text{next}$

$S.\text{code}$

    if  $c < d$  goto L1

    goto L2

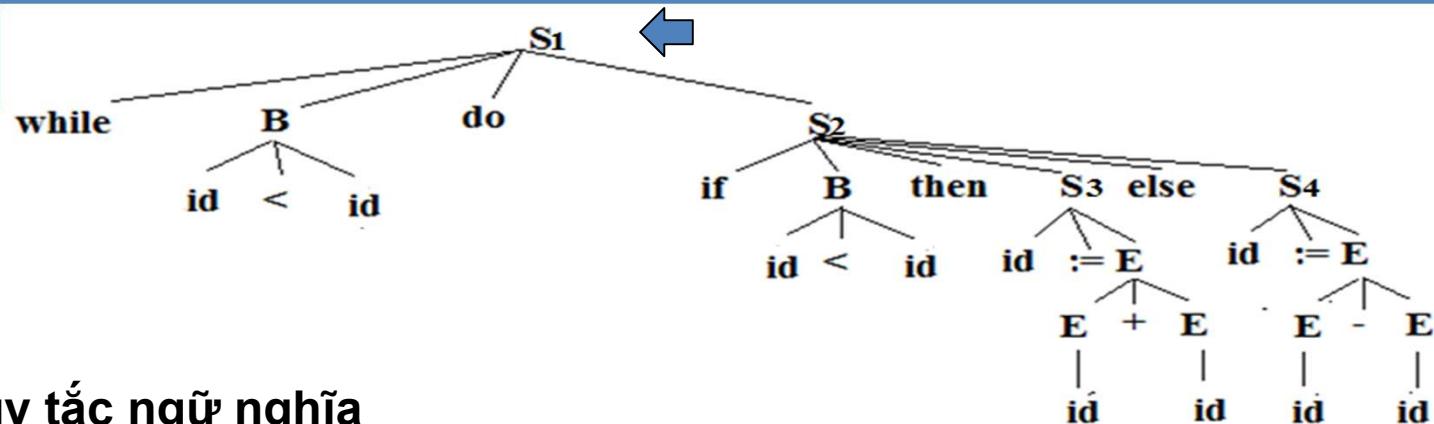
L1:  $t1 = y + z$

$x = t1$

    goto S.next

L2:  $t2 = y - z$

$x = t1$



## Quy tắc ngữ nghĩa

$S \rightarrow \text{while } B \text{ do } S_1$

```

begin = newlabel()
B.true = newlabel()
B.false = S.next
S1.next = begin
S.code = label(begin) || B.code
|| label(B.true) || S1.code
|| gen('goto' begin)
  
```

## Thuộc tính

```

begin = newlabel() -> L3
B.true = newlabel() -> L4
B.False = S.next
S1.next = begin = L3 => S2.next = S3.next = L3
S.code
L3: if a < b goto L4
    goto L0
L4: if c < d goto L1
    goto L2
L1: t1 = y + z
    x = t1
    goto L3
L2: t2 = y - z
    x = t1
    goto L3
  
```

Nhãn L0 sẽ xuất hiện trong chương trình khi sử dụng các quy tắc ngữ nghĩa của luật  $S \rightarrow S_1 S_2$

## 4.5. Tối ưu mã

- Yêu cầu
  - Chương trình sau khi tối ưu phải tương đương với chương trình ban đầu
  - Tốc độ thực hiện trung bình tăng
  - Hiệu quả đạt được tương xứng với công sức bỏ ra
- Có thể tối ưu mã vào lúc nào
  - Mã nguồn- do người lập trình (giải thuật)
  - Mã trung gian
  - Mã đích

# Tối ưu cục bộ

1. Kỹ thuật để cải tiến mã một cách cục bộ.
2. Một phương pháp để cải tiến chương trình đích bằng cách xem xét một dãy lệnh trong mã đích và thay thế chúng bằng những đoạn mã ngắn hơn và hiệu quả hơn

## Xu hướng chính

1. Loại bỏ lệnh dư thừa
2. Thông tin dòng điều khiển
3. Giản lược biểu thức đại số
4. Sử dụng các đặc trưng ngôn ngữ

# Tối ưu cục bộ

- **Tính toán biểu thức hằng**

`x := 32` trở thành `x := 64`

x := x + 32

- Mã không đến được

goto L2

`x := x + 1` ← Không cần

- **Tối ưu dòng điều khiển**

goto L1

## trở thành

goto L2

10

L1: goto L2 ← Không cần nếu không còn lệnh sau L2

# Tối ưu cục bộ

- **Giản lược biểu thức đại số**

$x := x + 0 \leftarrow$  Không cần

- **Mã chết**

$x := 32 \leftarrow x$  không được dùng trong  
những lệnh tiếp theo

$y := x + y \rightarrow y := y + 32$

- **Giảm chi phí tính toán**

$x := x * 2 \rightarrow x := x + x$

$\rightarrow x := x << 1$

# Tối ưu trong từng khối cơ sở

1. Loại bỏ biểu thức con chung
2. Tính giá trị hằng
3. Copy Propagation
4. Loại mã chết...

# Loại biểu thức con chung

$t1 = i + 1$

$t2 = b[t1]$

$t3 = i + 1$

$a[t3] = t2$

$t1 = i + 1$

$t2 = b[t1]$

$t3 = i + 1 \quad \leftarrow Không cần$

$a[t1] = t2$

# i là hằng

```
i = 4  
t1 = i+1  
t2 = b[t1]  
a[t1] = t2
```

```
i = 4  
t1 = 5  
t2 = b[t1]  
a[t1] = t2
```

```
i = 4  
t1 = 5  
t2 = b[5]  
a[5] = t2
```

Mã nhận được:

```
i = 4  
t2 = b[5]  
a[5] = t2
```

# Copy Propagation

- $\text{tmp2} = \text{tmp1} ;$
- $\text{tmp3} = \text{tmp2} * \text{tmp1} ;$
- $\text{tmp4} = \text{tmp3} ;$
- $\text{tmp5} = \text{tmp3} * \text{tmp2} ;$
- $\text{c} = \text{tmp5} + \text{tmp3} ;$
- $\text{tmp3} = \text{tmp1} * \text{tmp1} ;$
- $\text{tmp5} = \text{tmp3} * \text{tmp1} ;$
- $\text{c} = \text{tmp5} + \text{tmp3} ;$

# Tối ưu vòng đơn giản

- **Phương pháp**

Chuyển những đoạn mã bất biến ra ngoài vòng lặp.

**Ví dụ:**

while (i <= limit - 2)

Chuyển thành

t := limit - 2

while (i <= t)

# Tối ưu trên DAG

- Vấn đề cần quan tâm
  - Loại bỏ biểu thức con chung
  - Tính các biểu thức hằng
  - Loại mã chết
  - Loại những dư thừa cục bộ...
- Ứng dụng một phương pháp tối ưu dẫn đến việc tạo ra những đoạn mã có thể ứng dụng phương pháp tối ưu khác.

# Mã ba địa chỉ của Quick Sort

1	$i = m - 1$
2	$j = n$
3	$t_1 = 4 * n$
4	$v = a[t_1]$
5	$i = i + 1$
6	$t_2 = 4 * i$
7	$t_3 = a[t_2]$
8	if $t_3 < v$ goto (5)
9	$j = j - 1$
10	$t_4 = 4 * j$
11	$t_5 = a[t_4]$
12	if $t_5 > v$ goto (9)
13	if $i \geq j$ goto (23)
14	$t_6 = 4 * i$
15	$x = a[t_6]$

16	$t_7 = 4 * i$
17	$t_8 = 4 * j$
18	$t_9 = a[t_8]$
19	$a[t_7] = t_9$
20	$t_{10} = 4 * j$
21	$a[t_{10}] = x$
22	goto (5)
23	$t_{11} = 4 * i$
24	$x = a[t_{11}]$
25	$t_{12} = 4 * i$
26	$t_{13} = 4 * n$
27	$t_{14} = a[t_{13}]$
28	$a[t_{12}] = t_{14}$
29	$t_{15} = 4 * n$
30	$a[t_{15}] = x$

# Khối cơ bản (basic block)

Chuỗi các lệnh kế tiếp nhau trong đó dòng điều khiển đi vào lệnh đầu tiên của khối và ra ở lệnh cuối cùng của khối mà không bị dừng hoặc rẽ nhánh.

Ví dụ

$t1 := a * a$

$t2 := a * b$

$t3 := 2 * t2$

$t4 := t1 + t2$

$t5 := b * b$

$t6 := t4 + t5$

# Giải thuật phân chia các khối cơ bản

**Input:** Dãy lệnh ba địa chỉ.

**Output:** Danh sách các khối cơ bản với mã lệnh ba địa chỉ của từng khối

**Phương pháp:**

1. Xác định tập các lệnh đầu (leader), của từng khối cơ bản
  - i) Lệnh đầu tiên của chương trình là lệnh đầu.
  - ii) Bất kỳ lệnh nào là đích nhảy đến của các lệnh GOTO có hoặc không có điều kiện là lệnh đầu
  - iii) Bất kỳ lệnh nào đi sau lệnh GOTO có hoặc không có điều kiện là lệnh đầu
2. Với mỗi lệnh đầu, khối cơ bản bao gồm nó và tất cả các lệnh tiếp theo không phải là lệnh đầu hay lệnh kết thúc chương trình

# Ví dụ

```
(1) prod := 0
(2) i := 1
(3) t1 := 4 * i
(4) t2 := a[t1]
(5) t3 := 4 * i
(6) t4 := b[t3]
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i<=20 goto (3)
```

- Lệnh (1) là lệnh đầu theo quy tắc i,
- Lệnh (3) là lệnh đầu theo quy tắc ii
- Lệnh sau lệnh (12) là lệnh đầu theo quy tắc iii.
- Các lệnh (1)và (2) tạo nên khối cơ bản thứ nhất.
- Lệnh (3) đến (12) tạo nên khối cơ bản thứ hai.

# Xác định khối cô bản

1	$i = m - 1$
2	$j = n$
3	$t_1 = 4 * n$
4	$v = a[t_1]$
5	$i = i + 1$
6	$t_2 = 4 * i$
7	$t_3 = a[t_2]$
8	$\text{if } t_3 < v \text{ goto (5)}$
9	$j = j - 1$
10	$t_4 = 4 * j$
11	$t_5 = a[t_4]$
12	$\text{if } t_5 > v \text{ goto (9)}$
13	$\text{if } i \geq j \text{ goto (23)}$
14	$t_6 = 4 * i$
15	$x = a[t_6]$

16	$t_7 = 4 * i$
17	$t_8 = 4 * j$
18	$t_9 = a[t_8]$
19	$a[t_7] = t_9$
20	$t_{10} = 4 * j$
21	$a[t_{10}] = x$
22	$\text{goto (5)}$
23	$t_{11} = 4 * i$
24	$x = a[t_{11}]$
25	$t_{12} = 4 * i$
26	$t_{13} = 4 * n$
27	$t_{14} = a[t_{13}]$
28	$a[t_{12}] = t_{14}$
29	$t_{15} = 4 * n$
30	$a[t_{15}] = x$

# DAG

B<sub>1</sub>

```
i = m - 1
j = n
t1 = 4 * n
v = a[t1]
```

B<sub>2</sub>

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B<sub>3</sub>

```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B<sub>4</sub>

```
if i >= j goto B6
```

B<sub>5</sub>

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
```

B<sub>6</sub>

```
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x
```

# Loại biểu thức con chung

B<sub>1</sub>

```
i = m - 1
j = n
t1 = 4 * n
v = a[t1]
```

B<sub>2</sub>

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B<sub>3</sub>

```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B<sub>4</sub>

```
if i >= j goto B6
```

B<sub>5</sub>

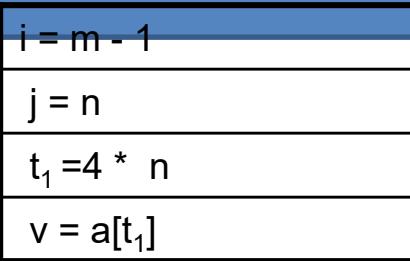
```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
```

B<sub>6</sub>

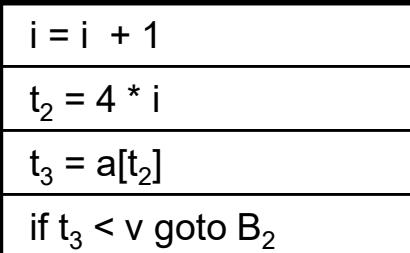
```
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x
```

# Loại biểu thức con chung

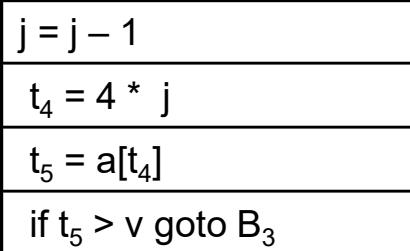
B<sub>1</sub>



B<sub>2</sub>



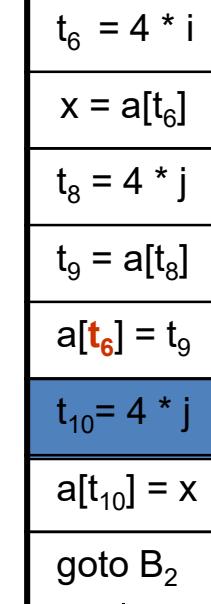
B<sub>3</sub>



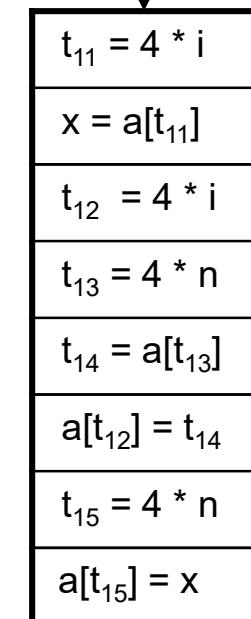
B<sub>4</sub>



B<sub>5</sub>

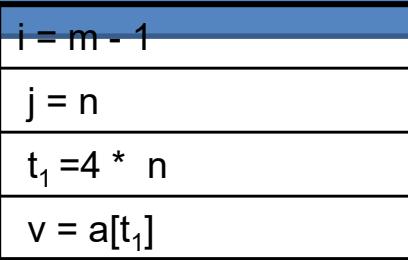


B<sub>6</sub>

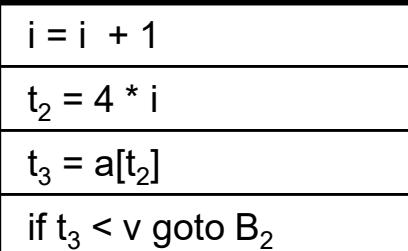


# Loại biểu thức con chung

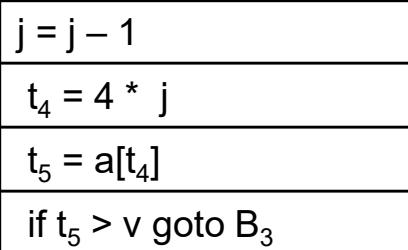
B<sub>1</sub>



B<sub>2</sub>



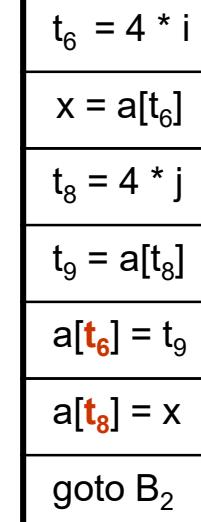
B<sub>3</sub>



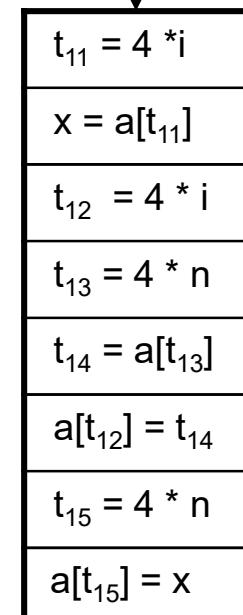
B<sub>4</sub>



B<sub>5</sub>

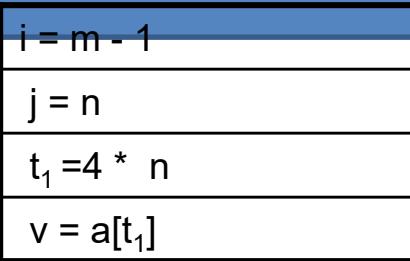


B<sub>6</sub>

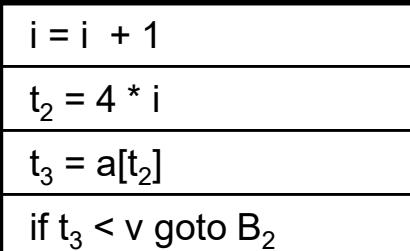


# Loại biểu thức con chung

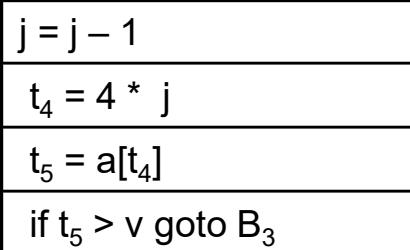
B<sub>1</sub>



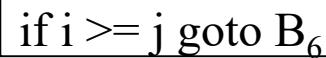
B<sub>2</sub>



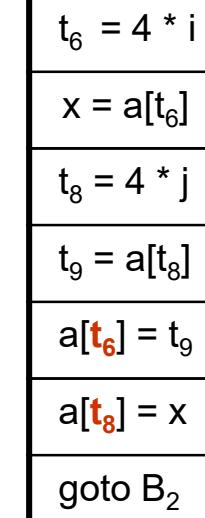
B<sub>3</sub>



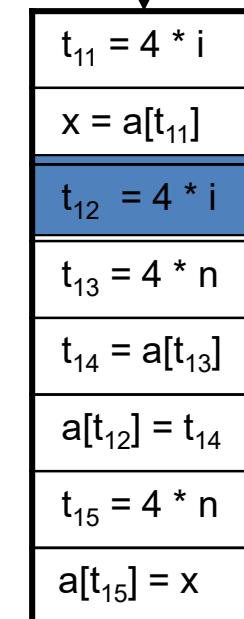
B<sub>4</sub>



B<sub>5</sub>

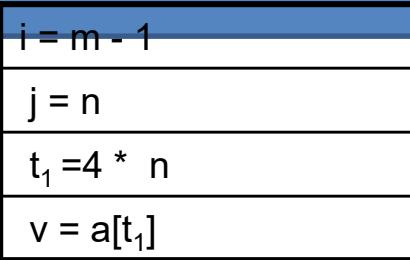


B<sub>6</sub>

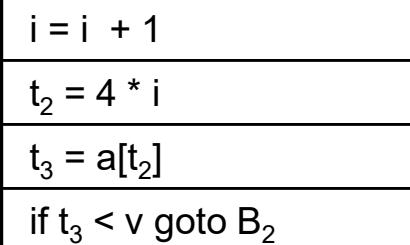


# Loại biểu thức con chung

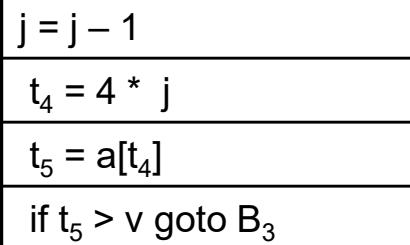
B<sub>1</sub>



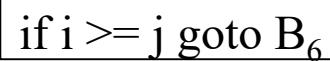
B<sub>2</sub>



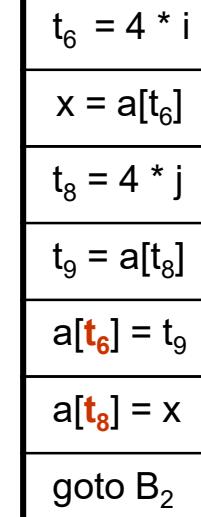
B<sub>3</sub>



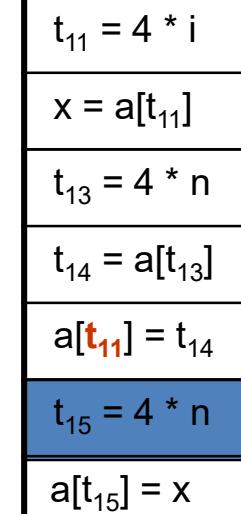
B<sub>4</sub>



B<sub>5</sub>



B<sub>6</sub>



# Loại biểu thức con chung

B<sub>1</sub>

```

i = m - 1
j = n
t1 = 4 * n
v = a[t1]

```

B<sub>2</sub>

```

i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2

```

B<sub>3</sub>

```

j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3

```

B<sub>4</sub>

```
if i >= j goto B6
```

B<sub>5</sub>

```

t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2

```

B<sub>6</sub>

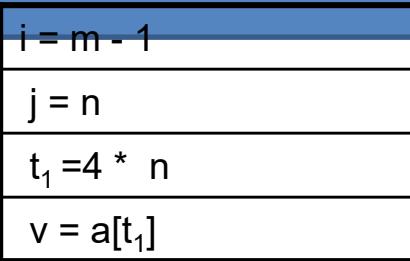
```

t11 = 4 * i
x = a[t11]
t13 = 4 * n
t14 = a[t13]
a[t11] = t14
a[t13] = x

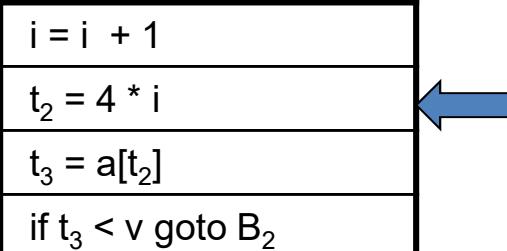
```

# Loại biểu thức con chung

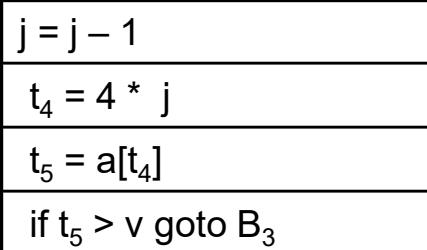
B<sub>1</sub>



B<sub>2</sub>



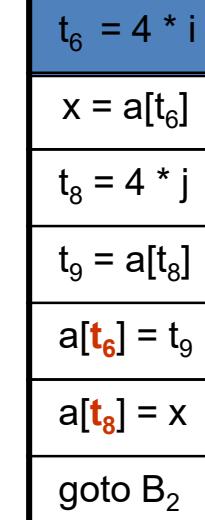
B<sub>3</sub>



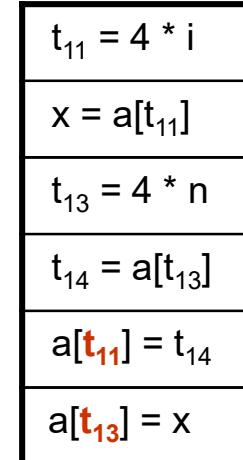
B<sub>4</sub>



B<sub>5</sub>

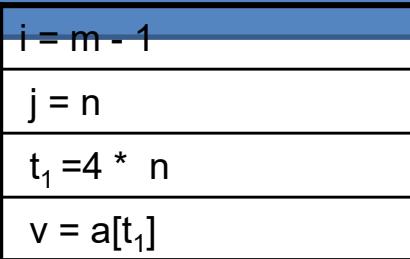


B<sub>6</sub>

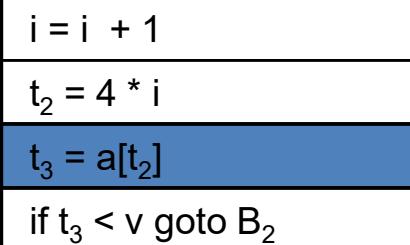


# Loại biểu thức con chung

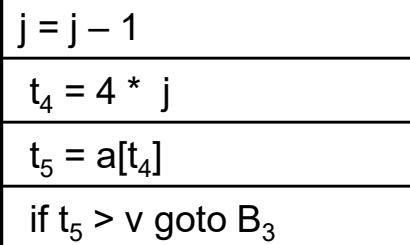
B<sub>1</sub>



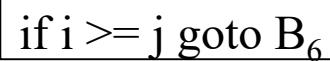
B<sub>2</sub>



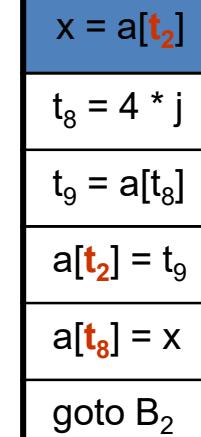
B<sub>3</sub>



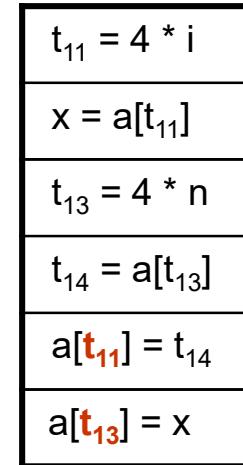
B<sub>4</sub>

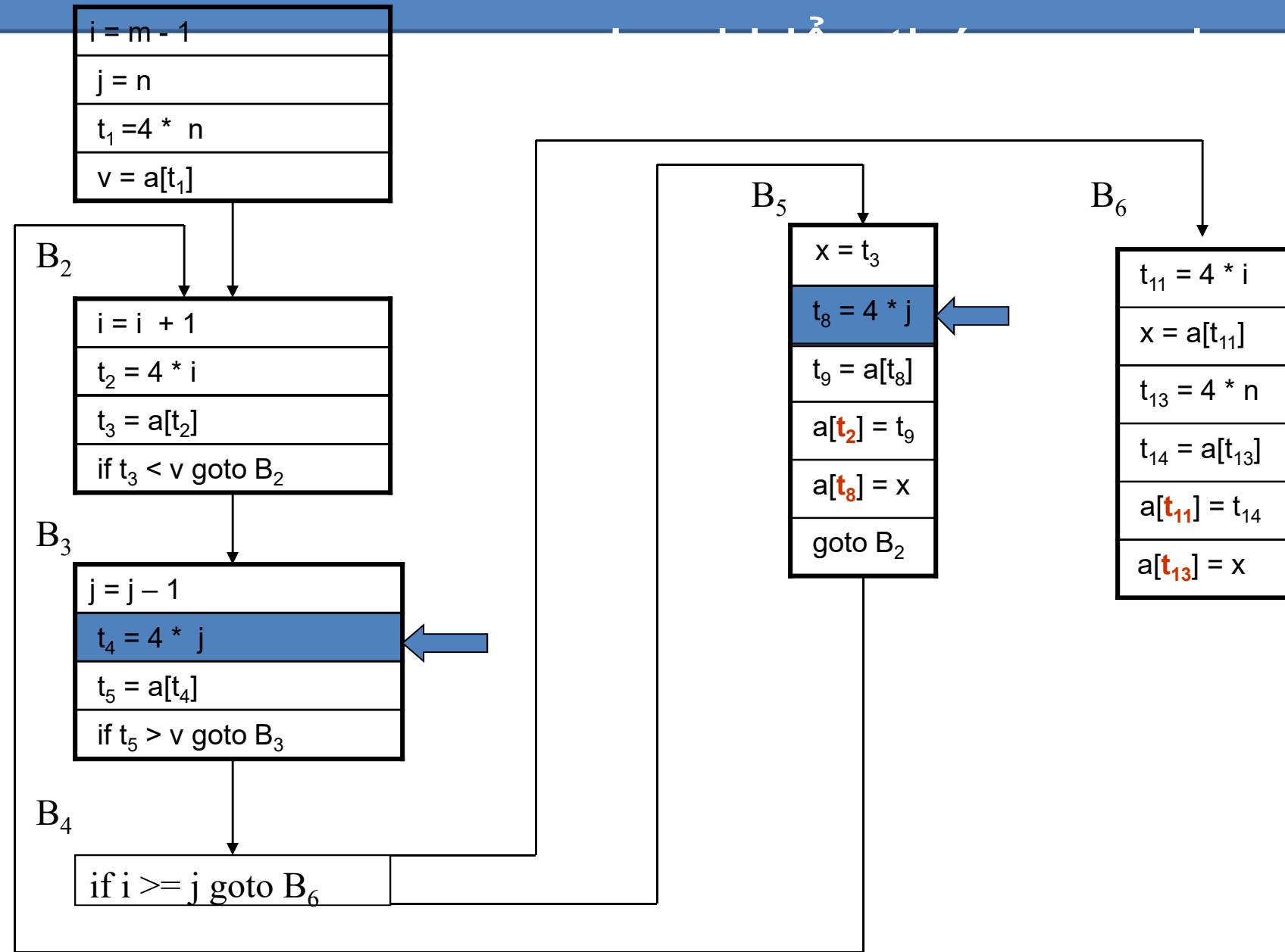


B<sub>5</sub>



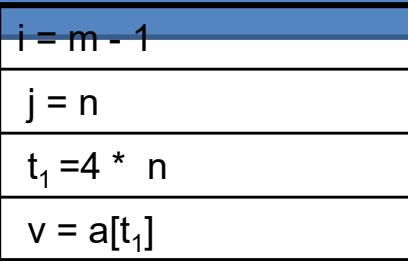
B<sub>6</sub>



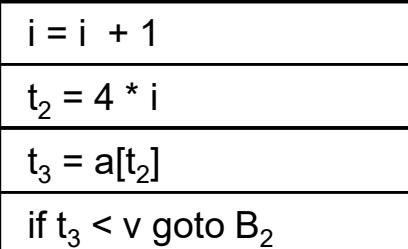


# Loại biểu thức con chung

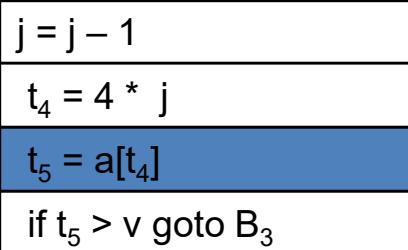
B<sub>1</sub>



B<sub>2</sub>



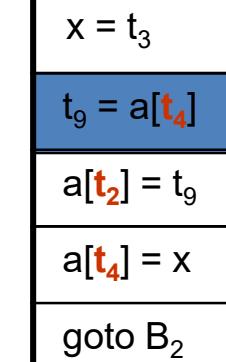
B<sub>3</sub>



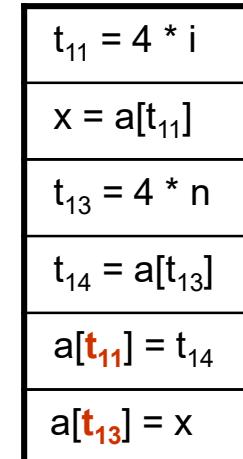
B<sub>4</sub>



B<sub>5</sub>

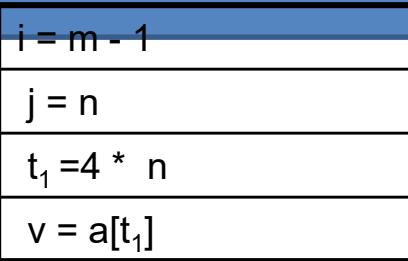


B<sub>6</sub>

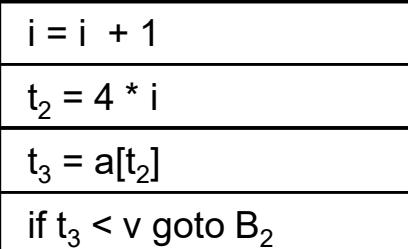


# Loại biểu thức con chung

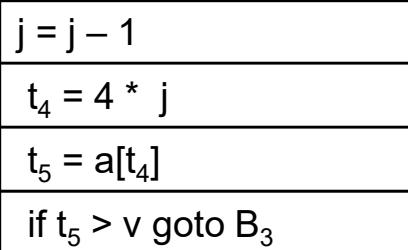
B<sub>1</sub>



B<sub>2</sub>



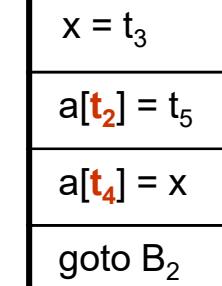
B<sub>3</sub>



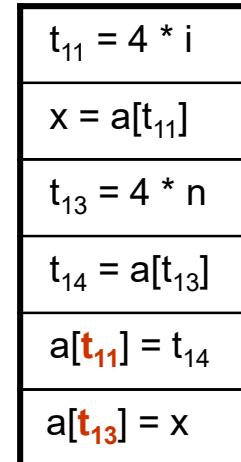
B<sub>4</sub>



B<sub>5</sub>

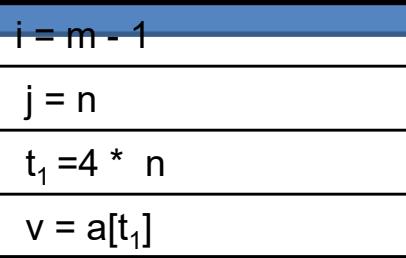


B<sub>6</sub>

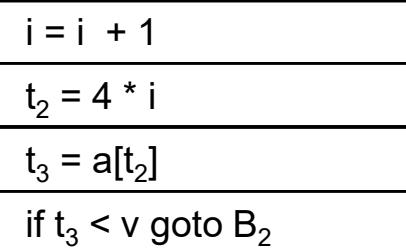


# Loại biểu thức con chung

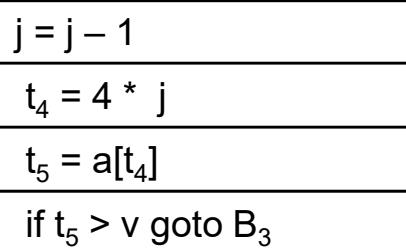
B<sub>1</sub>



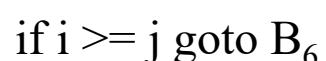
B<sub>2</sub>



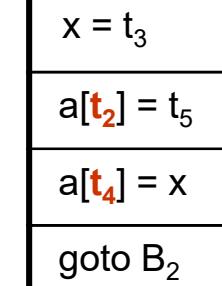
B<sub>3</sub>



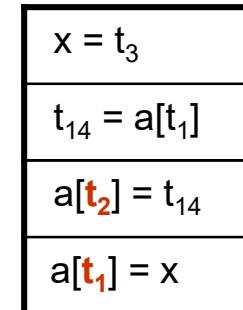
B<sub>4</sub>



B<sub>5</sub>



B<sub>6</sub>



Tương tự với B<sub>6</sub>

# Copy propagation

B<sub>1</sub>

```
i = m - 1
j = n
t1 = 4 * n
v = a[t1]
```

B<sub>2</sub>

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B<sub>3</sub>

```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B<sub>4</sub>

```
if i >= j goto B6
```

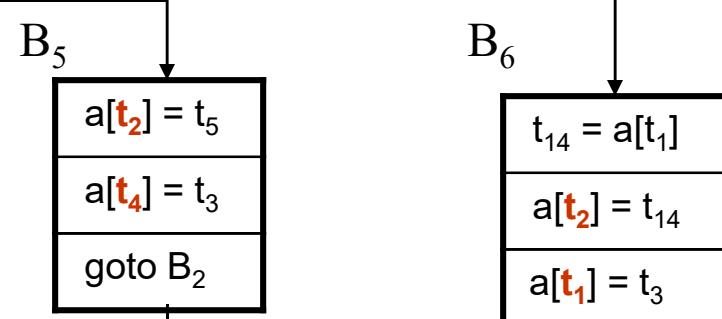
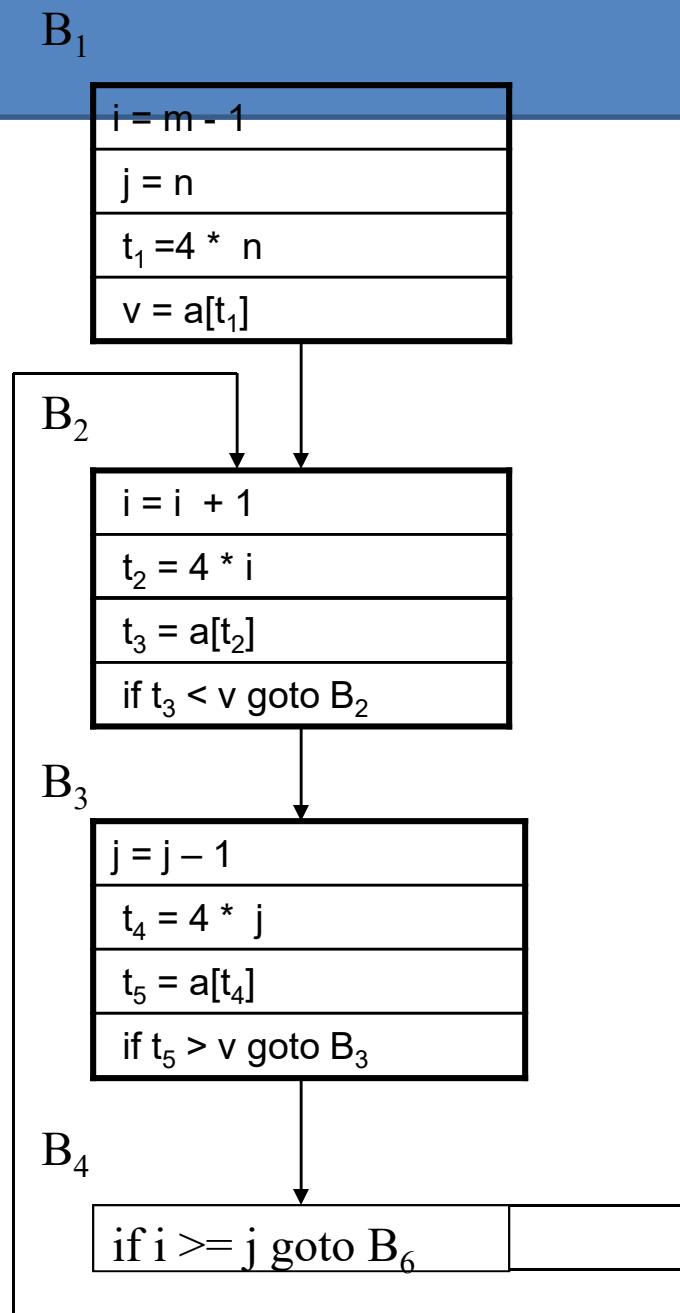
B<sub>5</sub>

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

B<sub>6</sub>

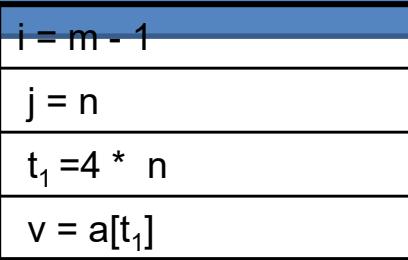
```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

# Copy propagation

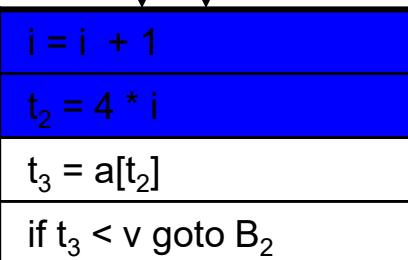


# Giảm chi phí

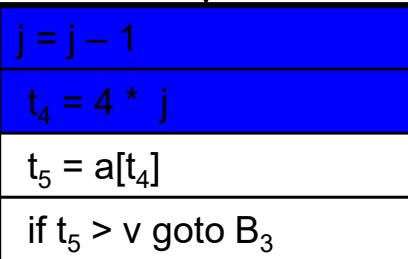
B<sub>1</sub>



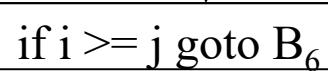
B<sub>2</sub>



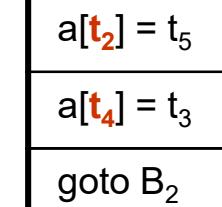
B<sub>3</sub>



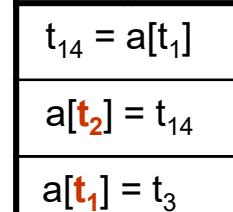
B<sub>4</sub>



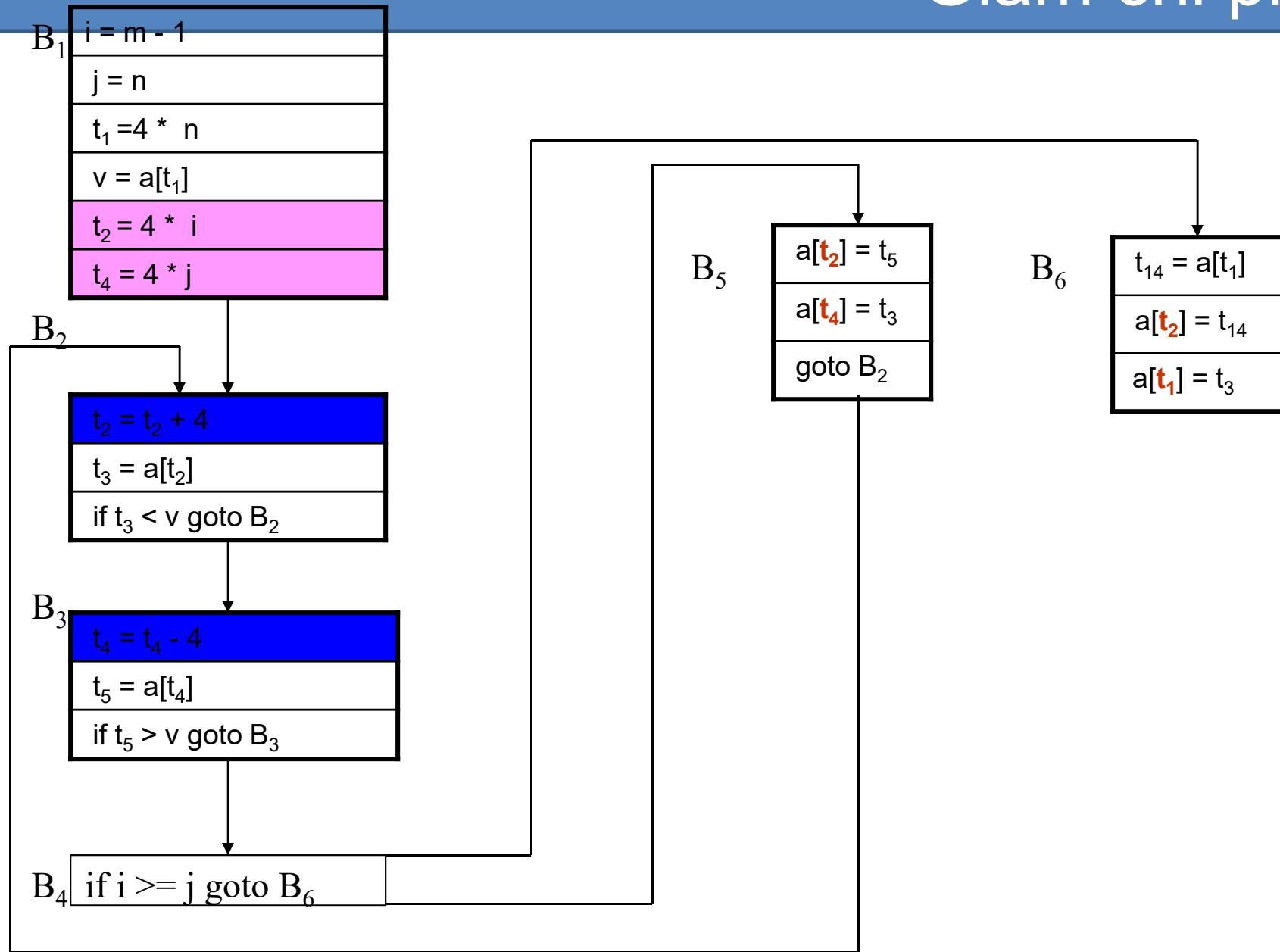
B<sub>5</sub>



B<sub>6</sub>



# Giảm chi phí



## 4.6. Sinh mã đích

- Tổng quan về sinh mã đích
- Máy ngăn xếp
  - Tổ chức bộ nhớ
  - Bộ lệnh
- Sinh mã cho các lệnh cơ bản
- Xây dựng bảng ký hiệu
  - Biến
  - Tham số
  - Hàm, thủ tục và chương trình

# Chương trình đích

- Viết trên một ngôn ngữ trung gian
- Là dạng Assembly của máy giả định (máy ảo)
- Máy ảo làm việc với bộ nhớ stack
- Việc thực hiện chương trình thông qua một interpreter
- Interpreter mô phỏng hành động của máy ảo thực hiện tập lệnh assembly của nó

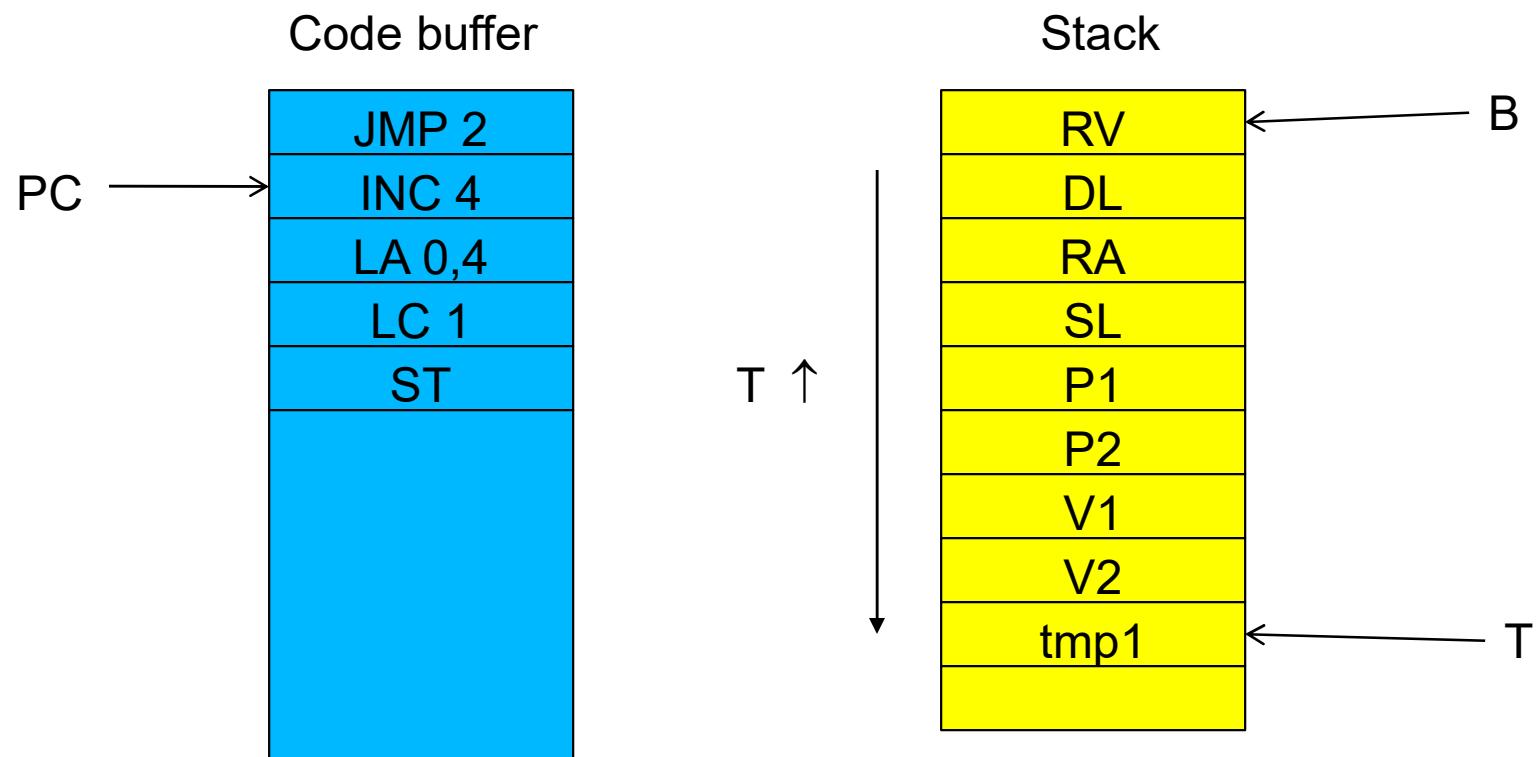
# Chương trình đích được dịch từ

- Mã nguồn
- Mã trung gian

# Máy ngăn xếp

- Máy ngăn xếp là một hệ thống tính toán
  - Sử dụng ngăn xếp để lưu trữ các kết quả trung gian của quá trình tính toán
  - Kiến trúc đơn giản
  - Bộ lệnh đơn giản
- Máy ngăn xếp có hai vùng bộ nhớ chính
  - Khối lệnh: chứa mã thực thi của chương trình
  - Ngăn xếp: sử dụng để lưu trữ các kết quả trung gian

# Máy ngăn xếp



# Máy ngăn xếp

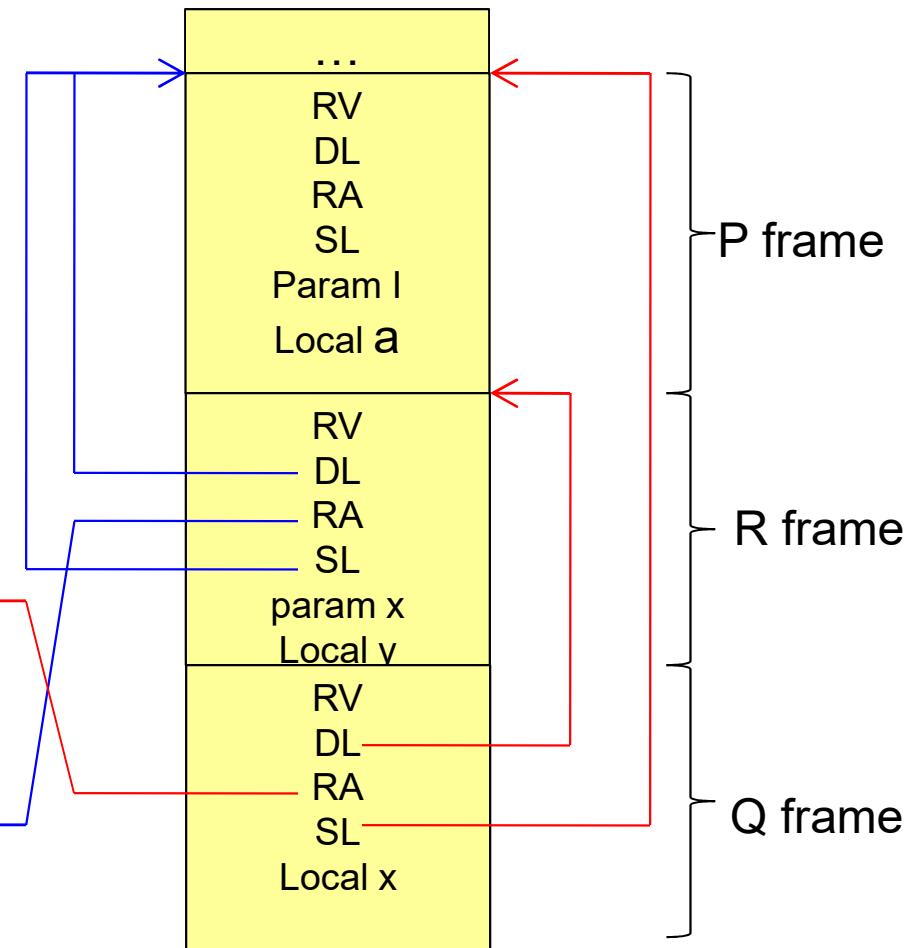
- **Thanh ghi**
  - PC (program counter): con trỏ lệnh trỏ tới lệnh hiện tại đang thực thi trên bộ đệm chương trình
  - B (base) : con trỏ trỏ tới địa chỉ gốc của vùng nhớ cục bộ. Các biến cục bộ được truy xuất gián tiếp qua con trỏ này
  - T (top); trỏ tới đỉnh của ngăn xếp

# Máy ngăn xếp

- Bản hoạt động (activation record/stack frame)
  - Không gian nhớ cấp phát cho mỗi chương trình con (hàm/thủ tục/chương trình chính) khi chúng được kích hoạt
    - Lưu giá trị tham số
    - Lưu giá trị biến cục bộ
    - Lưu các thông tin khác
      - Giá trị trả về của hàm – RV
      - Địa chỉ cơ sở của bản hoạt động của chương trình con gọi tới (caller) – DL
      - Địa chỉ lệnh quay về khi kết thúc chương trình con – RA
      - Địa chỉ cơ sở của bản hoạt động của chương trình con bao ngoài – SL
  - Một chương trình con có thể có nhiều bản hoạt động

# Máy ngăn xếp

```
Procedure P(I : integer);
  Var a : integer;
  Function Q;
    Var x : char;
    Begin
      ...
      return
    End;
  Procedure R(X: integer);
    Var y : char;
    Begin
      ...
      y = Call Q;
      ...
    End;
  Begin
    ...
    Call R(1);
    ...
  End;
...
```



# Máy ngăn xếp

- RV (return value): Lưu trữ giá trị trả về cho mỗi hàm
- DL (dynamic link): Sử dụng để hồi phục ngũ cảnh của chương trình gọi (caller) khi chương trình được gọi (callee) kết thúc
- RA (return address): Sử dụng để tìm tới lệnh tiếp theo của caller khi callee kết thúc
- SL (static link): Sử dụng để truy nhập các biến phi cục bộ

# Tập lệnh của máy ngăn xếp

- Bộ lệnh



LA	Load Address	$t := t + 1; \ s[t] := \text{base}(p) + q;$
LV	Load Value	$t := t + 1; \ s[t] := s[\text{base}(p) + q];$
LC	Load Constant	$t := t + 1; \ s[t] := q;$
LI	Load Indirect	$s[t] := s[s[t]]; \ t := t + 1;$
INT	Increment T	$t := t + q;$
DCT	Decrement T	$t := t - q;$

# Tập lệnh của máy ngăn xếp

- Bộ lệnh



J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];

# Tập lệnh của máy ngăn xếp

- Bộ lệnh



RC	Read Character	read one character into $s[s[t]]$ ; $t:=t-1$ ;
RI	Read Integer	read integer to $s[s[t]]$ ; $t:=t-1$ ;
WRC	Write Character	write one character from $s[t]$ ; $t:=t-1$ ;
WRI	Write Integer	write integer from $s[t]$ ; $t:=t-1$ ;
WLN	New Line	CR & LF

# Tập lệnh của máy ngăn xếp

- Bộ lệnh



AD	Add	$t := t - 1; \ s[t] := s[t] + s[t+1];$
SB	Subtract	$t := t - 1; \ s[t] := s[t] - s[t+1];$
ML	Multiply	$t := t - 1; \ s[t] := s[t] * s[t+1];$
DV	Divide	$t := t - 1; \ s[t] := s[t] / s[t+1];$
NEG	Negative	$s[t] := -s[t];$
CV	Copy Top of Stack	$s[t+1] := s[t]; \ t := t + 1;$

# Tập lệnh của máy ngăn xếp

- Bộ lệnh

op	p	q
----	---	---

EQ	Equal	$t := t - 1; \text{ if } s[t] = s[t+1] \text{ then } s[t] := 1 \text{ else } s[t] := 0;$
NE	Not Equal	$t := t - 1; \text{ if } s[t] \neq s[t+1] \text{ then } s[t] := 1 \text{ else } s[t] := 0;$
GT	Greater Than	$t := t - 1; \text{ if } s[t] > s[t+1] \text{ then } s[t] := 1 \text{ else } s[t] := 0;$
LT	Less Than	$t := t - 1; \text{ if } s[t] < s[t+1] \text{ then } s[t] := 1 \text{ else } s[t] := 0;$
GE	Greater or Equal	$t := t - 1; \text{ if } s[t] \geq s[t+1] \text{ then } s[t] := 1 \text{ else } s[t] := 0;$
LE	Less or Equal	$t := t - 1; \text{ if } s[t] \leq s[t+1] \text{ then } s[t] := 1 \text{ else } s[t] := 0;$

# Xây dựng bảng ký hiệu

- **Bổ sung thông tin cho biến**
  - Vị trí trên frame
  - Phạm vi
- **Bổ sung thông tin cho tham số**
  - Vị trí trên frame
  - Phạm vi
- **Bổ sung thông tin cho hàm/thủ tục/chương trình**
  - Địa chỉ bắt đầu
  - Kích thước của frame
  - Số lượng tham số của hàm/thủ tục

# Sinh mã lệnh gán

**V := exp**

```
<code of l-value v>    // đẩy địa chỉ của v lên stack
<code of exp>          // đẩy giá trị của exp lên stack
ST
```

# Sinh mã lệnh if

**If <dk> Then statement;**

```
<code of dk>      // đẩy giá trị điều kiện dk lên stack
FJ L
<code of statement>
L:
...
```

**If <dk> Then st1 Else st2;**

```
<code of dk>      // đẩy giá trị điều kiện dk lên stack
FJ L1
<code of st1>
J L2
L1:
<code of st2>
L2:
...
```

# Sinh mã lệnh while

## **While <dk> Do statement**

```
L1:  
  <code of dk>  
  FJ L2  
  <code of statement>  
  J L1  
L2:  
  ...
```

# Sinh mã lệnh for

**For v := exp1 to exp2 do statement**

```
<code of l-value v>
CV // nhân đôi địa chỉ của v
<code of exp1>
ST // lưu giá trị đầu của v
L1:
CV
LI // lấy giá trị của v
<code of exp2>
LE
FJ L2
<code of statement>
CV;CV;LI;LC 1;AD;ST; // Tăng v lên 1
J L1
L2:
DCT 1
...
```

# Lấy địa chỉ/giá trị biến

- Khi lấy địa chỉ/giá trị một biến cần tính đến phạm vi của biến
  - Biến cục bộ được lấy từ frame hiện tại
  - Biến phi cục bộ được lấy theo các StaticLink với cấp độ lấy theo “độ sâu” của phạm vi hiện tại so với phạm vi của biến

# Lấy địa chỉ của tham số hình thức

- Trong lệnh gán, khi **IValue** là tham số
- Cũng cần tính độ sâu như biến
  - Nếu là tham trị: địa chỉ cần lấy chính là **địa chỉ** của tham trị
  - Nếu là tham biến: vì giá trị của tham biến chính là **địa chỉ** muốn truy nhập, địa chỉ cần lấy chính là **giá trị** của tham biến.

## Lấy giá trị của tham số hình thức

- Khi tính toán giá trị của **Factor**
- Cũng cần tính độ sâu như biến
  - Nếu là tham trị: giá trị của tham trị chính là giá trị cần lấy.
  - Nếu là tham biến: giá trị của tham số là địa chỉ của giá trị cần lấy.

# Lấy địa chỉ của giá trị trả về của hàm

- Giá trị trả về luôn nằm ở offset 0 trên frame
- Chỉ cần tính độ sâu giống như với biến hay tham số hình thức

# Sinh lời gọi hàm/thủ tục

- Lời gọi
  - Hàm gấp trong sinh mã cho **factor**
  - Thủ tục gấp trong sinh mã lệnh **Callst**
- Trước khi sinh lời gọi hàm/thủ tục cần phải nạp giá trị cho các tham số hình thức bằng cách
  - Tăng giá trị T lên 4 (bỏ qua RV,DL,RA,SL)
  - Sinh mã cho k tham số thực sự
  - Giảm giá trị T đi 4 + k
  - Sinh lệnh CALL

# Sinh mã cho lệnh CALL (p, q)

**CALL (p, q)**

s [t+2] :=b ;	<i>// Lưu lại dynamic link</i>
s [t+3] :=pc ;	<i>// Lưu lại return address</i>
s [t+4] :=base (p) ;	<i>// Lưu lại static link</i>
b :=t+1 ;	<i>// Base mới và return value</i>
pc :=q ;	<i>// địa chỉ lệnh mới</i>

Giả sử cần sinh lệnh CALL cho hàm/thủ tục A

Lệnh CALL(p, q) có hai tham số:

- p: Độ sâu của lệnh CALL, chứa static link.  
Base(p) = base của frame chương trình con chứa khai báo của A.
- q: Địa chỉ lệnh mới  
q + 1 = địa chỉ đầu tiên của dãy lệnh cần thực hiện khi gọi A.

# Hoạt động khi thực hiện lệnh CALL(p, q)

1. Điều khiển `pc` chuyển đến địa chỉ bắt đầu của chương trình con /\* `pc = p` \*/
2. `pc` tăng thêm 1 /\* `pc ++` \*/
3. Lệnh đầu tiên thông thường là lệnh nhảy `J` để bỏ qua mã lệnh của các khai báo hàm/ thủ tục cục bộ trên `code buffer`.
4. Lệnh tiếp theo là lệnh `INT` tăng `T` đúng bằng kích thước frame để bỏ qua frame chứa vùng nhớ của các tham số và biến cục bộ.

# Hoạt động khi thực hiện lệnh CALL(p, q)

5. Thực hiện các lệnh và stack biến đổi tương ứng.
6. Khi kết thúc
  1. Thủ tục (lệnh `EP`): toàn bộ frame được giải phóng, con trỏ `T` đặt lên đỉnh frame cũ.
  2. Hàm (lệnh `EF`): frame được giải phóng, chỉ chứa giá trị trả về tại `offset 0`, con trỏ `T` đặt lên đầu frame hiện thời (`offset 0`).

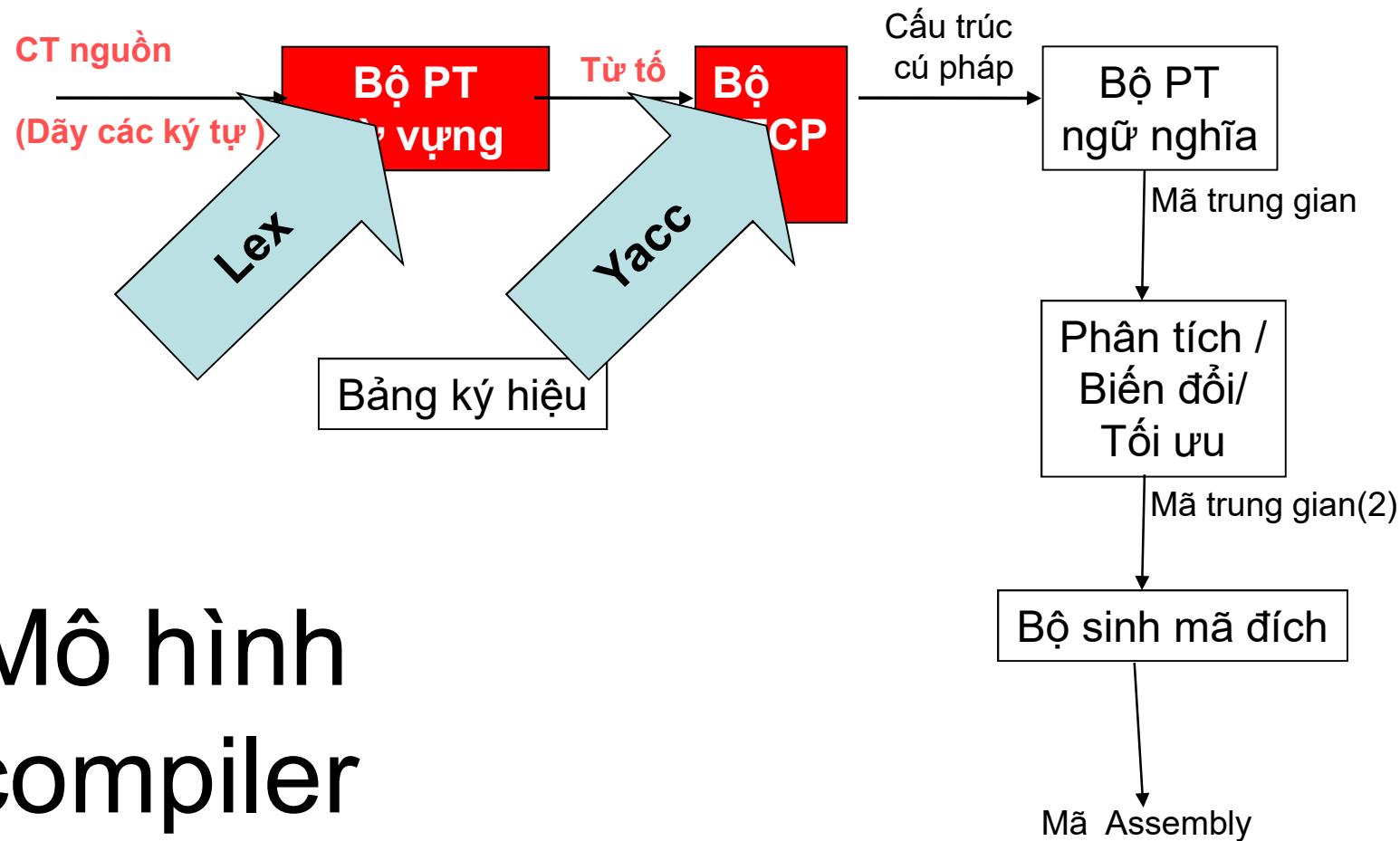
## Sinh mã đích từ mã ba địa chỉ

- Bộ sinh mã trung gian đưa ra mã ba địa chỉ
- Tối ưu trên mã ba địa chỉ
- Từ mã ba địa chỉ đã tối ưu sinh ra mã đích phù hợp với một mô tả máy ảo

# Chương 5: Bộ sinh Compiler

- 5.1. Nguyên lý
- 5.2. Lex và Flex
- 5.3. YACC và BISON

## 5.1. Nguyên lý



Mô hình  
compiler

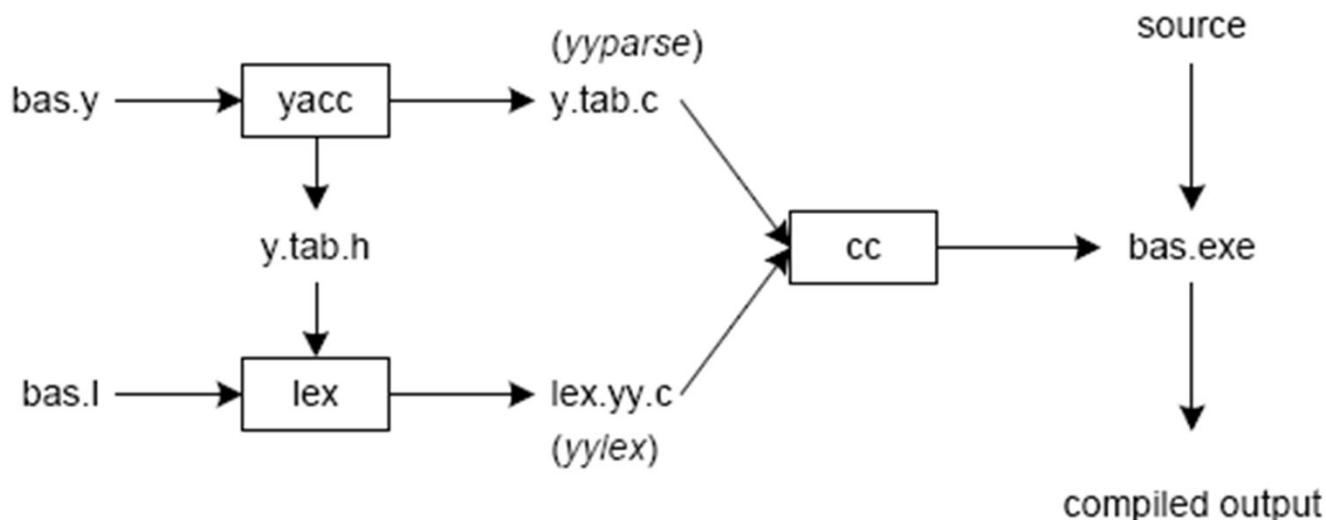
# Lex và Yacc

- **Lex**

- Sinh mã lệnh C cho bộ phân tích từ vựng
- Từ tố được mô tả bằng biểu thức chính quy

- **Yacc**

- Sinh mã lệnh C cho bộ phân tích cú pháp LR(1)
- Văn phạm biểu diễn bằng BNF



## 5.2. Lex và Flex

- Công cụ để đặc tả bộ phân tích từ vựng cho nhiều ngôn ngữ
- Một số bộ sinh phân tích từ vựng
  - **Lex và Yacc của AT &T**
  - **Lexer trong ANTLR** (ANother Tool for Language Recognition) ĐH San Francisco
  - **Flex của Berkeley Lab**

# Biểu thức chính quy (regular expression)

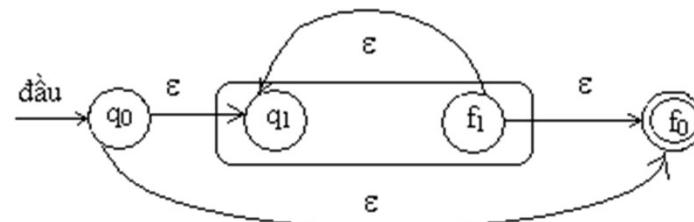
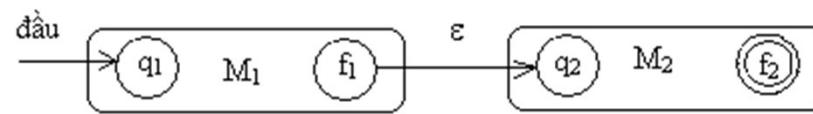
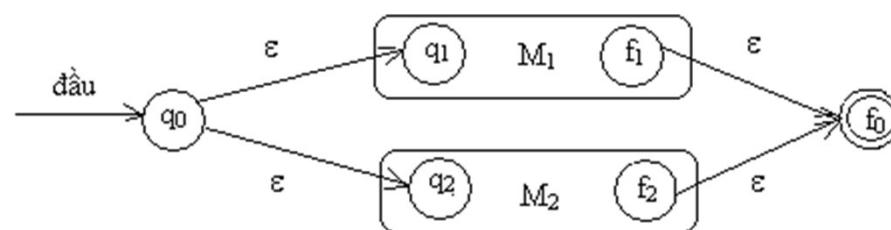
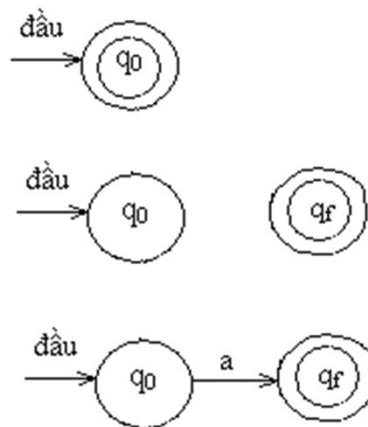
Cho  $\Sigma$  là một bảng chữ.

- $\emptyset$  là biểu thức chính quy biểu diễn tập  $\emptyset$
- $\epsilon$  là biểu thức chính quy biểu diễn tập  $\{\epsilon\}$
- $\forall a \in \Sigma$ ,  $a$  là biểu thức chính quy biểu diễn tập  $\{a\}$
- Nếu  $r$  và  $s$  là các biểu thức chính quy biểu diễn các tập  $R$  và  $S$  tương ứng thì  $(r + s)$ ,  $(rs)$ ,  $(r^*)$  là các biểu thức chính quy biểu diễn các tập  $R \cup S$ ,  $RS$  và  $R^*$  tương ứng.

## Ví dụ

- Tập các xâu trên  $\{0, 1\}$  bắt đầu bằng 1  
$$(1((0+1)^*))$$
- Tập các xâu trên  $\{a,b\}$  có chứa 3 ký hiệu a liên tiếp  
$$((((((a + b)^*)a)a)a)((a+b)^*))$$

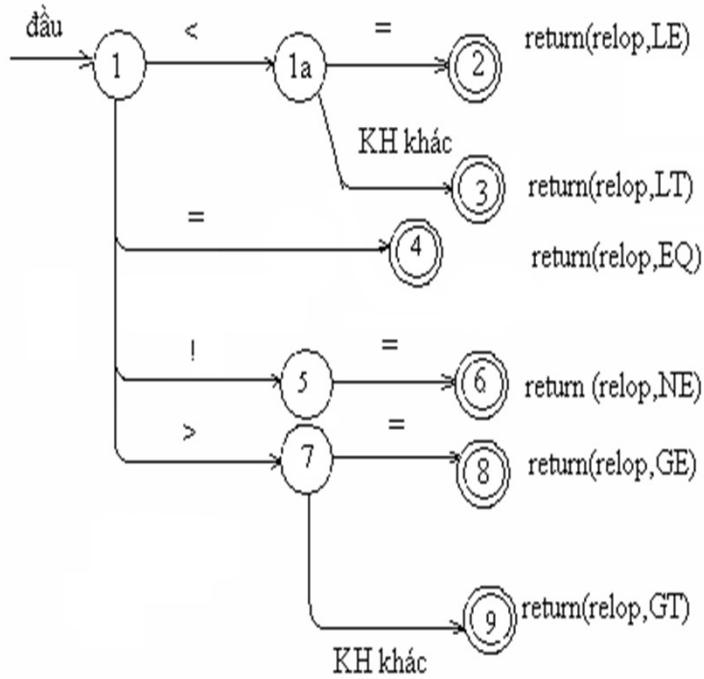
# Tương đương giữa biểu thức chính quy và ôtômat hữu hạn



# Biểu diễn hình thức ôtômat hữu hạn

- $M = (\Sigma, Q, \delta, q_0, F)$ ,
- $\Sigma$  : Bảng chữ của xâu vào
- $Q$  : Tập hữu hạn trạng thái
- $q_0 \in Q$  : Trạng thái đầu
- $F \subseteq Q$  : Tập trạng thái kết thúc
- $\delta : \Sigma \times Q \rightarrow Q$  gọi là hàm chuyển trạng thái

# Tùy ô tô mat hữu hạn sang bộ phân tích từ vựng

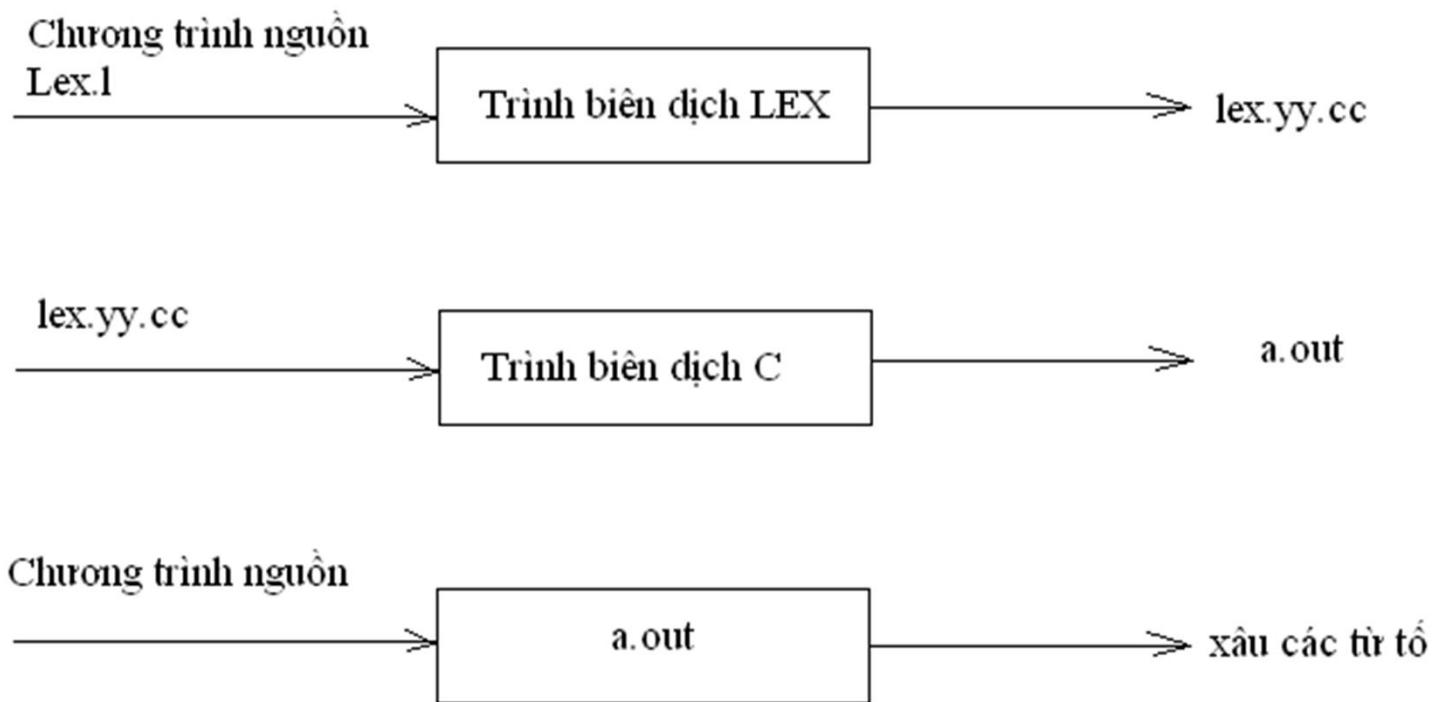


```
case 0 : c= nextchar();
    if(c==blank || c== =tab || c== =newline ){
        state = 0;
        lexeme_beginning++;
        //chuyển con trỏ đến đầu từ tố
    }
case 1:
    if(c== '=' <') state = 1a;
    else if (c== '=') state = 4;
    else if (c== '!') state = 5;
    else if (c== '>') state = 7;
    else state =fail();break;
case 1a: c:=nextchar();
if (c== '=') state = 2;
Else state=3;
case 2: return (leq)
```

# Bộ sinh phân tích từ vựng

- Vào
  - Đặc tả từ vựng của ngôn ngữ lập trình (định nghĩa chính quy)
- Ra
  - Chương trình phân tích từ vựng (chương trình C)

# Cấu trúc của bộ sinh phân tích từ vựng



## Đặc tả

- Đặc tả bằng một chương trình Lex.l viết bằng ngôn ngữ Lex.
- Chương trình dịch của Lex sẽ sinh ra một chương trình C lex.yy.c.
- Lex.yy.c được cho chạy qua một trình biên dịch C, sinh ra a.out.
- a.out chính là bộ phân tích từ vựng của ngôn ngữ đưa vào

## Chương trình Lex.yy.c

- Biểu diễn hình thức của một ôtômat hữu hạn, xây dựng từ các biểu thức chính quy được mô tả trong tệp lex.l ,
- Chứa các thủ tục chuẩn dùng bảng d để nhận dạng các từ tố.

# Đặc tả Lex

*Phần khai báo*

*%*

*Các luật dịch*

*%*

*Các thủ tục bổ trợ*

# Phần khai báo

- Biến
- Hằng hiện
- Định nghĩa chính quy

# Các luật dịch

$p_1$  {hành động 1}

$p_2$  {hành động 2}

.....

$p_3$  {hành động 3}

# Ví du

```
%{  
enum yytokentype {  
NUMBER = 258,  
ADD = 259,  
SUB = 260,  
MUL = 261,  
DIV = 262,  
ABS = 263,  
EOL = 264  
};  
int yylval;  
%}  
%%  
"+" { return ADD; }  
"-" { return SUB; }  
"**" { return MUL; }  
"/" { return DIV; }  
"|" { return ABS; }
```

```
[0-9]+ { yylval = atoi(yytext); return  
NUMBER; }  
\n { return EOL; }  
[ \t] { /* ignore whitespace */ }  
. { printf("Mystery character %c\n",  
*yytext); }  
%%  
main(int argc, char **argv)  
{  
int tok;  
while(tok = yylex()) {  
printf("%d", tok);  
if(tok == NUMBER) printf(" = %d\n",  
yylval);  
else printf("\n");  
}  
}
```

# Kết quả chạy thử nghiệm tệp a.out

<b>flex fb1-4.l</b>	<b>262</b>
<b>\$ cc lex.yy.c -lfl</b>	<b>258 = 34</b>
<b>\$ ./a.out</b>	<b>259</b>
<b>a / 34 +  45</b>	<b>263</b>
<b>Mystery character a</b>	<b>258 = 45</b>
	<b>264</b>

# Ví dụ: Dùng Flex sinh scanner cho KPL

- **Dựa trên các đặc điểm từ vựng của KPL**
  - Số nguyên: không dấu
  - Định danh : dùng chữ cái, chữ số, bắt đầu bằng chữ cái
  - Hằng ký tự
  - Bỏ qua ký tự trắng, chú thích
- **Xây dựng mô tả từ vựng scanner.I**
- **Dùng Flex sinh ra tệp lex.yy.c**
- **Thực hiện với ví dụ**

# Mô tả tệp scanner.l

## Phần 1: định nghĩa và hàm thư viện :

Tạo ra hàm yyloc() tính số dòng, số cột của từ tố

Tham số truyền vào là text , x, y. Trong đó :

Text: từ tố cần phân tích x số của dòng chứa từ tố

Y: số của cột chứa từ tố.

```
1 #option DAWWWRAP
2 ${
3
4     void yyloc(char* text, int *x, int *y){
5         while (*text != '\0'){
6             if (*text == '\n'){
7                 *y += 1;
8                 *x = 1;
9             } else {
10                 *x += 1;
11             }
12             text++;
13         }
14     }
15
16     int x = 1;
17     int y = 1;
18 }
19
20
21 NUMBER          [0-9]+
22 DELIMITER        [ \n\t\r]
23 CHAR             \[[\[:print:]\]]\'
24 IDENT            [a-zA-Z][a-zA-Z0-9]*
25 COMMENT          \(\*([^\*]|(\*+[^\*]))*\*+\)
26 ERROR            [^\*-/,:();]=a-zA-Z0-9<>
27
28 }
```

# Mô tả tệp scanner.l

## Phần 1: định nghĩa và hàm thư viện.

Định nghĩa chính quy:

- NUMBER : các số từ 0 → 9 ( có thể mở rộng ra gồm nhiều chữ số ) . Biểu thức chính quy : [0-9]+
- DELIMITER : Các dấu trắng , khoảng cách , xuống dòng. Biểu thức chính quy : [ \n\t\r]
- CHAR : Nhận diện các kí tự bắt đầu bởi dấu ' , kết thúc bởi dấu ' và in ra phần bên trong 2 dấu nháy ([:print:]) tương đương với hành động in ra ) . Biểu thức chính quy : '\[[\[:print:]]\]'
- IDENT: các biến được định nghĩa . Cho phép các kí tự hoặc số. Biểu thức chính quy : [a-zA-Z][a-zA-Z0-9]\*
- COMMENT: khớp với các kí tự được bắt đầu bằng "(" và kết thúc bởi ")". Biểu thức chính quy : \(\(([^\*]|([\*+[^\*]]))\*)\)\*\(+\)
- ERROR : khi các kí tự không nằm trong khoảng cho phép ( automat kpl) Biểu thức chính quy : [^+\-\*/;.:()]=a-zA-Z0-9<>]

```
1 #option DAWYKAR
2 ${
3
4     void xyloc(char* text, int *x, int *y){
5         while (*text != '\0'){
6             if (*text == '\n'){
7                 *y += 1;
8                 *x = 1;
9             } else {
10                 *x += 1;
11             }
12             text++;
13         }
14     }
15
16     int x = 1;
17     int y = 1;
18 }
19
20
21 NUMBER           [0-9] +
22 DELIMITER        [ \n\t\r]
23 CHAR             '\[[\[:print:]]\]'
24 IDENT            [a-zA-Z][a-zA-Z0-9] *
25 COMMENT          \(\(([^*]|([*+[^*]]))*)\)*\(+\)
26 ERROR            [^+\-*/;.:()]=a-zA-Z0-9<>
27
28 }
```

# Mô tả tệp scanner.l

- Phần quy tắc dịch :**  
Do cấu trúc của flex dịch từ trên xuống , do đó ta sẽ ưu tiên thư tự cho các biểu thức cho phù hợp. Ví dụ như EOF hay COMMENT sẽ duyệt đầu tiên và in ra .  
Hàm phải được thể hiện trên 1 dòng , ta sẽ in ra chữ tương ứng với cú pháp được đưa vào từ input , cộng thêm địa chỉ của từ được phân tích bằng cách sử dụng hàm lấy vị trí đã nêu ở trên.
- Chương trình sẽ in ra bao gồm vị trí của từ “x-y” , phân tích cú pháp từ .

```
DELIMITER      [\n\t\r]
CHAR          \\\{\(print\)\}\\
IDENT         [a-zA-Z][a-zA-Z0-9]*
COMMENT        \\((\\*\\((\\*\\|\\*\\+\\*\\))\\*\\+\\*\\))\\*\\+\\*\\)
ERROR          [^+\\-*/..:\\]=a-zA-Z0-9<>]

;;
<<EOF>>      {return 0;}
(COMMENT)      {vylloc(vytext, sx, sy);}
(DELIMITER)    {vylloc(vytext, sx, sy);}

"+"
"-"
"*"
"/"
";"
","
";"
":"
"(""
")"
"!="
">="
"!="
"="
"<"
">"
":"
"("
")"

"PROGRAM"
"CONST"
"TYPE"
"VAR"
"INTEGER"
"CHAR"

{printf(vyout, "#d:#s\n", y, x, "SB_PLUS"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_MINUS"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_TIMES"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_SLASH"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_SEMICOLON"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_COMMA"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_PERIOD"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_ASSIGN"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_LSEL"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_RSEL"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_NEQ"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_GE"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_LT"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_EQ"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_COLON"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_LPAR"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "SB_RPAR"); vylloc(vytext, sx, sy);}

{printf(vyout, "#d:#s\n", y, x, "KW_PROGRAM"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "KW_CONST"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "KW_TYPE"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "KW_VAR"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "KW_INTEGER"); vylloc(vytext, sx, sy);}
{printf(vyout, "#d:#s\n", y, x, "KW_CHAR"); vylloc(vytext, sx, sy);}
```

# Mô tả tệp scanner.l

- **Phần thủ tục hỗ trợ**
- Cho phép file đọc input và in ra file output .

Khi run file ta sẽ gồm 3 phần là tên chương trình khi biên dịch ra file exe , tên file input , tên output (tùy chọn) .

Nếu sai cú pháp thoát chương trình , chương trình sẽ tự động in ra cách dùng như hình và thoát .

```
83 int main(argc, argv)
84 int argc;
85 char **argv;
86 {
87     extern FILE* yyin;
88     extern FILE* yyout;
89
90     if (argc == 3) {
91         yyin = fopen(argv[1], "r");
92         yyout = fopen(argv[2], "w");
93     } else {
94         printf("Usage: scanner.exe [input_file] [output_file]");
95         exit(1);
96     }
97     yylex();
98     return 0;
99 }
```

# 7. Phân tích file LEX.YY.C

- Đầu tiên chương trình sẽ khởi tạo các hàm định nghĩa và tên các thư viện cần phải sử dụng .
- Sau đó thiết lập các bộ nhớ đệm để phân tích cũng như lưu tạm biến.

```
lexyy.c
15  #ifndef c_plusplus
16  #ifndef __cplusplus
17  #define __cplusplus
18  #endif
19  #endif
20
21
22  #ifndef __cplusplus
23
24  #include <stdlib.h>
25  #include <unistd.h>
26
27  /* Use prototypes in function declarations. */
28  #define YY_USE_PROTOS
29
30  /* The "const" storage-class-modifier is valid. */
31  #define YY_USE_CONST
32
33  #else /* ! __cplusplus */
34
35  #if __STDC__
36
37  #define YY_USE_PROTOS
38  #define YY_USE_CONST
39
40  #endif /* __STDC__ */
41  #endif /* ! __cplusplus */
42
43  #ifdef __TURBOC__
44  #pragma warn -rcb
45  #pragma warn -use
46  #include <iob.h>
47  #include <stdlib.h>
48  #define YY_USE_CONST
49  #define YY_USE_PROTOS
50  #endif
51
```

```
lexyy.c
144 struct yy_buffer_state
145 {
146     FILE *yy_input_file;
147
148     char *yy_ch_buf;      /* input buffer */
149     char *yy_buf_pos;     /* current position in input buffer */
150
151     /* Size of input buffer in bytes, not including room for EOF
152     * characters.
153     */
154     yy_size_t yy_buf_size;
155
156     /* Number of characters read into yy_ch_buf, not including EOF
157     * characters.
158     */
159     int yy_n_chars;
160
161     /* Whether we "own" the buffer - i.e., we know we created it,
162     * and can realloc() it to grow it, and should free() it to
163     * delete it.
164     */
165     int yy_is_our_buffer;
166
167     /* Whether this is an "interactive" input source; if so, and
168     * if we're using stdio for input, then we want to use getc()
169     * instead of fread(), to make sure we stop fetching input after
170     * each newline.
171     */
172     int yy_is_interactive;
173
174     /* Whether we're considered to be at the beginning of a line.
175     * If so, '^' rules will be active on the next match, otherwise
176     * not.
177     */
178     int yy_at_bol;
179
180     /* Whether to try to fill the input buffer when we reach the
```

## 7. Phân tích file LEX.YY.C

- Tiếp theo , chương trình sẽ thiết kế các hàm để cài đặt cho yytext() truyền vào :
  - Đưa ra hàm để cấp phát bộ nhớ , tạo các mảng lưu biến , hàm dùng để khớp đầu vào các pattern và lọc các đầu vào .
  - Tạo cây pattern duyệt theo thứ tự trong scanner.l dùng cấu trúc switch case . Ở đây, các input sẽ được match đúng với yêu cầu và .. . . . .

```
lex.y.c
695     goto yy_find_action;
696
697 case YY_STATE_EOF(INITIAL):
698 #line 30 "scanner3.l"
699 {return 0;}
700     YY_BREAK
701 case 1:
702 YY_RULE_SETUP
703 #line 31 "scanner3.l"
704 {yyloc(yytext, &x, &y);}
705     YY_BREAK
706 case 2:
707 YY_RULE_SETUP
708 #line 32 "scanner3.l"
709 {yyloc(yytext, &x, &y);}
710     YY_BREAK
711 case 3:
712 YY_RULE_SETUP
713 #line 34 "scanner3.l"
714 {fprintf(yyout, "%d-%d:%s\n", y, x, "SB_PLUS"); yyloc(yytext, &x, &y);}
715     YY_BREAK
716 case 4:
717 YY_RULE_SETUP
718 #line 35 "scanner3.l"
719 {fprintf(yyout, "%d-%d:%s\n", y, x, "SB_MINUS"); yyloc(yytext, &x, &y);}
720     YY_BREAK
721 case 5:
722 YY_RULE_SETUP
723 #line 36 "scanner3.l"
724 {fprintf(yyout, "%d-%d:%s\n", y, x, "SB_TIMES");}
725     YY_BREAK
726 case 6:
727 YY_RULE_SETUP
728 #line 37 "scanner3.l"
729 {fprintf(yyout, "%d-%d:%s\n", y, x, "SB_SLASH");}
730     YY_BREAK
731 case 7:
```

## 7. Phân tích file LEX.YY.C

- Cuối cùng là bộ phân tích để lấy các Input đầu vào .

Chương trình sẽ xây dựng các hàm vào ra , thêm và xóa bộ đệm

.....

Và cuối cùng là hàm main giống với hàm main đã set trong file scanner.l với chức năng tương đương.

```
1804 static void yy_flex_free( void *ptr )
1805 #else
1806 static void yy_flex_free( ptr )
1807 void *ptr;
1808 #endif
1809 {
1810     free( ptr );
1811 }
1812
1813 #if YY_MAIN
1814 int main()
1815 {
1816     yylex();
1817     return 0;
1818 }
1819 #endif
1820 #line 81 "scanner3.l"
1821
1822
1823 int main(argc, argv)
1824 int argc;
1825 char **argv;
1826 {
1827     extern FILE* yyin;
1828     extern FILE* yyout;
1829
1830     if (argc == 3) {
1831         yyin = fopen(argv[1], "r");
1832         yyout = fopen(argv[2], "w");
1833     } else {
1834         printf("Usage: scanner.exe [input_file] [output_file]");
1835         exit(1);
1836     }
1837     yylex();
1838     return 0;
1839 }
```

## 5.3. YACC / BISON

- YACC: “Yet another Compiler-Compiler”
- Nhập vào văn phạm phi ngũ cảnh, tự động xây dựng bảng phân tích LALR.
- Sử dụng BISON:
  - Tệp đưa vào: grammar.y
  - Tệp đưa ra: grammar.tab.c
  - Hàm chính đọc từng dòng, thực hiện từng luật.

# Phân tích LR

- Phân tích SLR( $k$ )
  - Xem xét nhiều từ tố để gạt và thông tin follow của nhiều từ tố để thu gọn
- Phân tích LR(1) tổng quát
  - Chứa các ký hiệu xem trước trong quá trình xây dựng ô tô mat hữu hạn
- Phân tích LALR(1)
  - Làm đơn giản sơ đồ trạng thái của văn phạm LR(1)
  - Được YACC / Bison sử dụng

# Cấu trúc một file BISON

Các định nghĩa bao gồm cả mã trực tiếp viết  
trong %{ %}

%%

Các hành động và các luật của văn phạm  
tương ứng

%%

Mã bổ sung như là:

main(){ return yyparse(); }

## Ví dụ: Tính toán biểu thức

- Các luật mô tả văn phạm :
  - $S' \rightarrow \text{exp}$
  - $\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$
  - $\text{term} \rightarrow \text{term} * \text{factor} \mid \text{factor}$
  - $\text{factor} \rightarrow \text{NUMBER} \mid ( \text{exp} )$
- Các hành động kết hợp để thực hiện các tính toán số học

# Cú pháp các luật của BISON

- Về trái :
- Mỗi lựa chọn có một hành động đi kèm sau đó là |
- ; sau hành động cuối cùng
- Ví dụ

```
factor :   NUMBER { $$ = $1; }
           |   '(' exp ')' { $$ = $2; }
           ;
```

# Các hành động của BISON

- Các luật chứa hành động được viết trong { } (mã)
- Các biến khai báo trước:
  - \$\$ giá trị kết quả của luật (YYSTYPE hoặc int)
  - \$1 giá trị từ tố đầu, \$2 giá trị từ tố thứ hai vv...
- Ví dụ
  - Exp: exp '+' term {\$\$ = \$1 + \$3;}

# Kết hợp BISON và FLEX

- Định nghĩa token ở mục định nghĩa:
  - %token ID <giá trị>
  - Các giá trị được chọn > 256
- Đảm bảo lex.yy.c và yy.tab.c thống nhất trên các định nghĩa ID
  - #define ID val
- Dịch đồng thời cả hai công cụ
  - g++ -o myparser yy.tab.c lex.yy.c -lfl

# FLEX trong BISON

- Mỗi luật cần trả về một loại từ tố
  - Ví dụ trả về NUMBER;
- Giá trị của một từ tố có thể được lưu trong biến toàn cục yylval
  - Ví dụ yylval = myAtol(yytext);

# Quản lý lỗi

- Lỗi = Ô trống trong bảng phân tích
- Để đưa ra đúng
  - Tìm lỗi trước khi thu gọn, nếu có thể