

# CHƯƠNG I

## GIỚI THIỆU VỀ SỰ BIÊN DỊCH

### Nội dung chính:

Để máy tính có thể hiểu và thực thi một chương trình được viết bằng ngôn ngữ cấp cao, ta cần phải có một *trình biên dịch* thực hiện việc chuyển đổi chương trình đó sang chương trình ở dạng ngôn ngữ đích. Chương này trình bày một cách tổng quan về *cấu trúc của một trình biên dịch* và mối liên hệ giữa nó với các thành phần khác - “hàng” của nó - như bộ tiền xử lý, bộ tải và soạn thảo liên kết, v.v. Cấu trúc của trình biên dịch được mô tả trong chương là một cấu trúc mức quan niệm bao gồm các giai đoạn: Phân tích từ vựng, Phân tích cú pháp, Phân tích ngữ nghĩa, Sinh mã trung gian, Tối ưu mã và Sinh mã đích.

### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải nắm được một cách tổng quan về nhiệm vụ của các thành phần của một trình biên dịch, mối liên hệ giữa các thành phần đó và môi trường nơi trình biên dịch thực hiện công việc của nó.

### Tài liệu tham khảo:

- [1] **Trình Biên Dịch** - Phan Thị Tươi (Trường Đại học kỹ thuật Tp.HCM) - NXB Giáo dục, 1998.
- [2] **Compilers : Principles, Technique and Tools** - Alfred V.Aho, Jeffrey D.Ullman - Addison - Wesley Publishing Company, 1986.
- [3] **Compiler Design** – Reinhard Wilhelm, Dieter Maurer - Addison - Wesley Publishing Company, 1996.

## I. TRÌNH BIÊN DỊCH

Nói một cách đơn giản, trình biên dịch là một chương trình làm nhiệm vụ đọc một chương trình được viết bằng một ngôn ngữ - *ngôn ngữ nguồn* (source language) - rồi dịch nó thành một chương trình tương đương ở một ngôn ngữ khác - *ngôn ngữ đích* (target language). Một phần quan trọng trong quá trình dịch là ghi nhận lại các lỗi có trong chương trình nguồn để thông báo lại cho người viết chương trình.

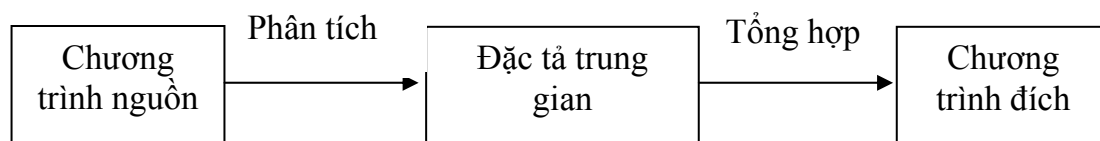


Hình 1.1 - Một trình biên dịch

### 1. Mô hình phân tích - tổng hợp của một trình biên dịch

Chương trình dịch thường bao gồm hai quá trình : phân tích và tổng hợp

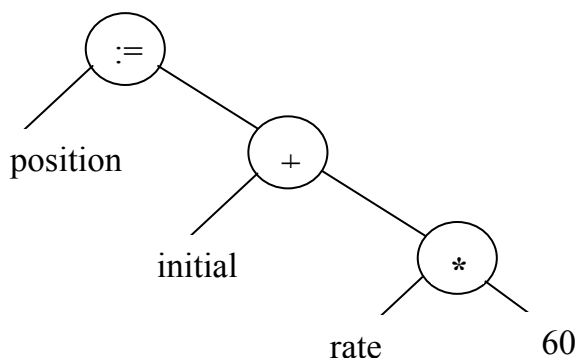
- Phân tích → đặc tả trung gian
- Tổng hợp → chương trình đích



**Hình 1.2 - Mô hình phân tích - tổng hợp**

Trong quá trình phân tích chương trình nguồn sẽ được phân rã thành một cấu trúc phân cấp, thường là dạng cây - *cây cú pháp* (syntax tree) mà trong đó có mỗi nút là một toán tử và các nhánh con là các toán hạng.

**Ví dụ 1.1:** Cây cú pháp cho câu lệnh gán `position := initial + rate * 60`



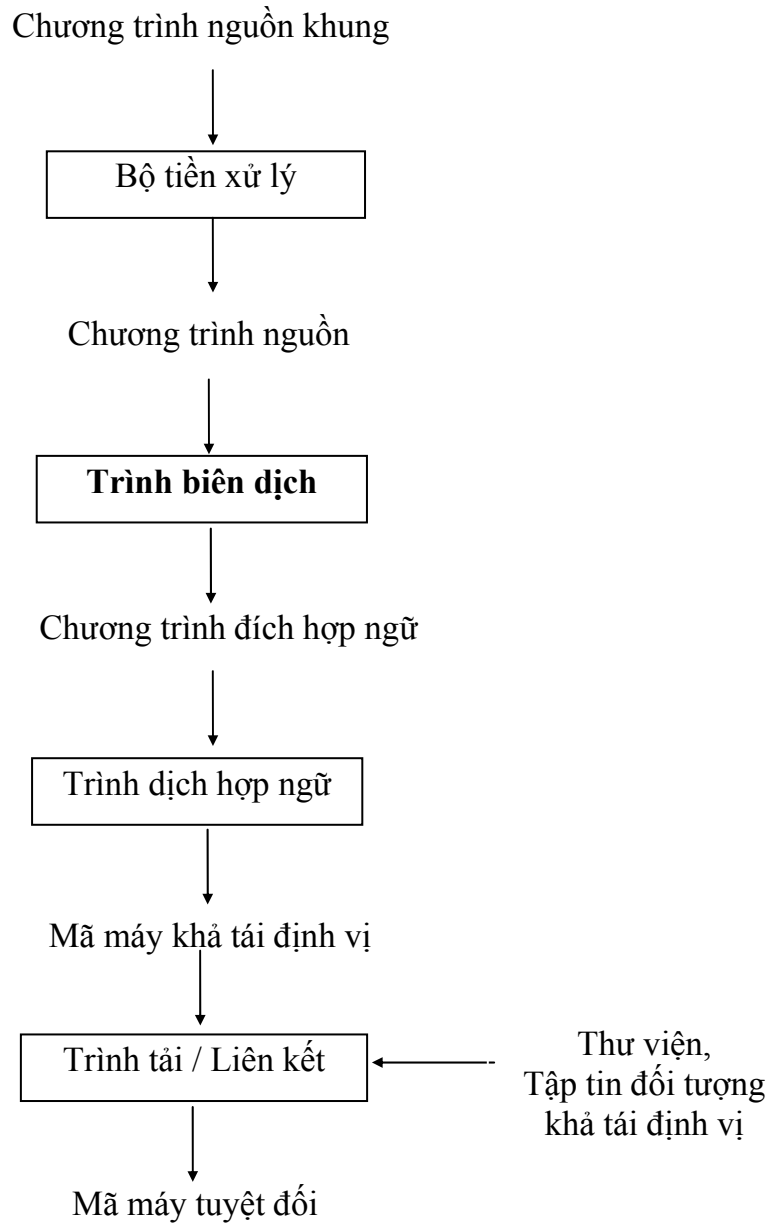
## 2. Môi trường của trình biên dịch

Ngoài trình biên dịch, chúng ta có thể cần dùng nhiều chương trình khác nữa để tạo ra một chương trình đích có thể thực thi được (executable). Các chương trình đó gồm: Bộ tiền xử lý, Trình dịch hợp ngữ, Bộ tải và soạn thảo liên kết.

Một chương trình nguồn có thể được phân thành các module và được lưu trong các tập tin riêng rẽ. Công việc tập hợp lại các tập tin này thường được giao cho một chương trình riêng biệt gọi là *bộ tiền xử lý* (preprocessor). Bộ tiền xử lý có thể "bung" các ký hiệu tắt được gọi là các macro thành các câu lệnh của ngôn ngữ nguồn.

Ngoài ra, chương trình đích được tạo ra bởi trình biên dịch có thể cần phải được xử lý thêm trước khi chúng có thể chạy được. Thông thường, trình biên dịch chỉ tạo ra mã lệnh hợp ngữ (assembly code) để *trình dịch hợp ngữ* (assembler) dịch thành dạng mã máy rồi được liên kết với một số thủ tục trong thư viện hệ thống thành các mã thực thi được trên máy.

Hình sau trình bày một quá trình biên dịch điển hình :



**Hình 1.3** - Một trình xử lý ngôn ngữ điển hình

## II. SỰ PHÂN TÍCH CHƯƠNG TRÌNH NGUỒN

Phần này giới thiệu về các quá trình phân tích và cách dùng nó thông qua một số ngôn ngữ định dạng văn bản.

### 1. Phân tích từ vựng (Lexical Analysis)

Trong một trình biên dịch, giai đoạn phân tích từ vựng sẽ đọc chương trình nguồn từ trái sang phải (quét nguyên liệu - scanning) để tách ra thành các thẻ từ (token).

**Ví dụ 1.2:** Quá trình phân tích từ vựng cho câu lệnh gán `position := initial + rate * 60` sẽ tách thành các token như sau:

1. Danh biểu `position`
2. Ký hiệu phép gán `:=`
3. Danh biểu `initial`

4. Ký hiệu phép cộng (+)
5. Danh biểu rate
6. Ký hiệu phép nhân (\*)
7. Số 60

Trong quá trình phân tích từ vựng các khoảng trắng (blank) sẽ bị bỏ qua.

## 2. Phân tích cú pháp (Syntax Analysis)

Giai đoạn phân tích cú pháp thực hiện công việc nhóm các thẻ từ của chương trình nguồn thành các *ngữ đoạn văn phạm* (grammatical phrase), mà sau đó sẽ được trình biên dịch tổng hợp ra thành phẩm. Thông thường, các ngữ đoạn văn phạm này được biểu diễn bằng dạng *cây phân tích cú pháp* (parse tree) với :

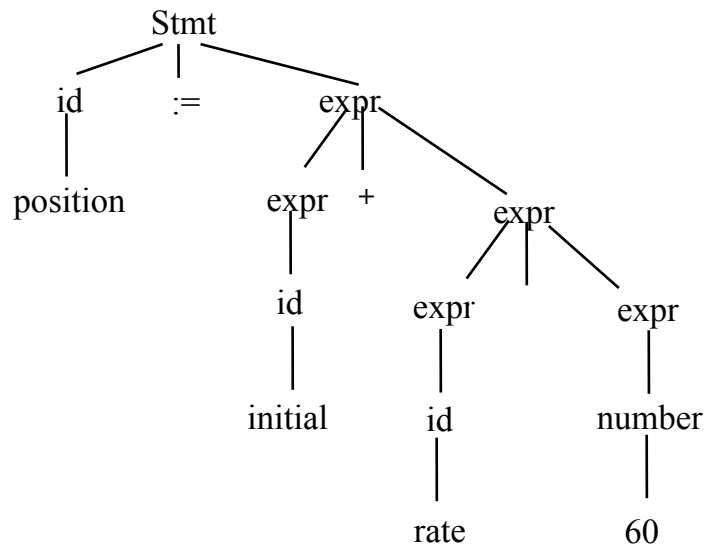
- Ngôn ngữ được đặc tả bởi các luật sinh.
- Phân tích cú pháp dựa vào luật sinh để xây dựng cây phân tích cú pháp.

**Ví dụ 1.3:** Giả sử ngôn ngữ đặc tả bởi các luật sinh sau :

$\text{Stmt} \rightarrow \text{id} := \text{expr}$

$\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid \text{id} \mid \text{number}$

Với câu nhập: position := initial + rate \* 60, cây phân tích cú pháp được xây dựng như sau :



**Hình 1.4** - Một cây phân tích cú pháp

Cấu trúc phân cấp của một chương trình thường được diễn tả bởi quy luật đệ qui.

**Ví dụ 1.4:**

- 1) Danh biểu (identifier) là một biểu thức (expr).
- 2) Số (number) là một biểu thức.
- 3) Nếu expr1 và expr2 là các biểu thức thì:

expr1 + expr2

expr1 \* expr2

(expr)

cũng là những biểu thức.

Câu lệnh (statement) cũng có thể định nghĩa đệ quy :

1) Nếu id1 là một danh biểu và expr2 là một biểu thức thì  $\text{id1} := \text{expr2}$  là một lệnh (stmt).

2) Nếu expr1 là một biểu thức và stmt2 là một lệnh thì

while (expr1) do stmt2

if (expr1) then stmt2

đều là các lệnh.

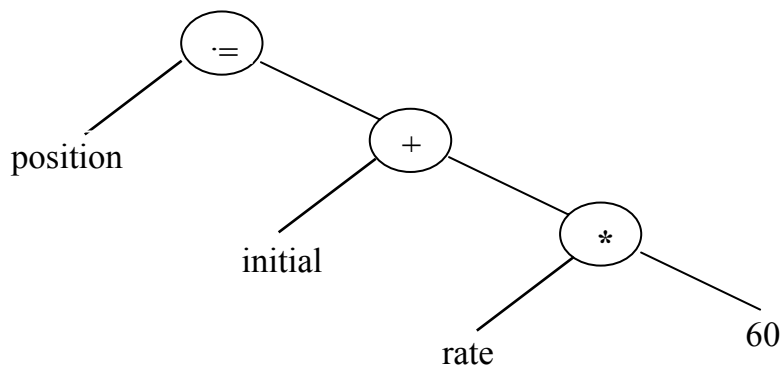
Người ta dùng các qui tắc đệ quy như trên để đặc tả luật sinh (production) cho ngôn ngữ. Sự phân chia giữa quá trình phân tích từ vựng và phân tích cú pháp cũng tùy theo công việc thực hiện.

### 3. Phân tích ngữ nghĩa (Semantic Analysis)

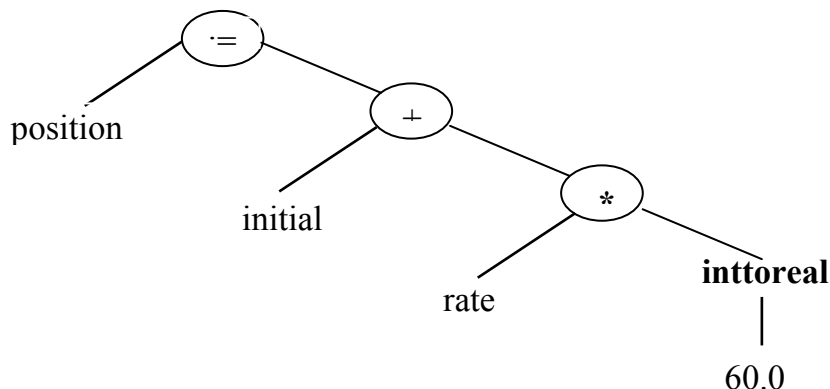
Giai đoạn phân tích ngữ nghĩa sẽ thực hiện việc kiểm tra xem chương trình nguồn có chứa lỗi về ngữ nghĩa hay không và tập hợp thông tin về kiểu cho giai đoạn sinh mã về sau. Một phần quan trọng trong giai đoạn phân tích ngữ nghĩa là *kiểm tra kiểu* (type checking) và ép chuyển đổi kiểu.

**Ví dụ 1.5:** Trong biểu thức  $\text{position} := \text{initial} + \text{rate} * 60$

Các danh biểu (tên biến) được khai báo là real, 60 là số integer vì vậy trình biên dịch đổi số nguyên 60 thành số thực 60.0



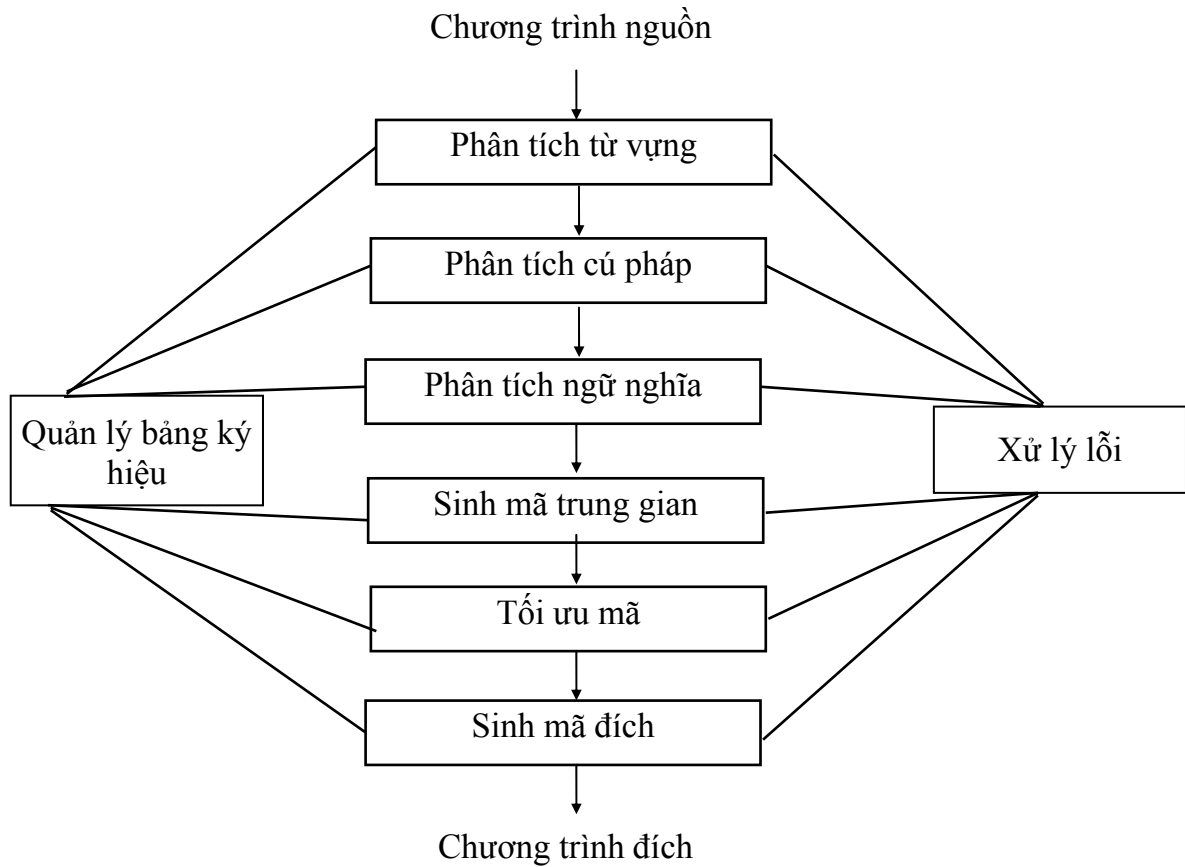
thành



**Hình 1.5** - Chuyển đổi kiểu trên cây phân tích cú pháp

### III. CÁC GIAI ĐOẠN BIÊN DỊCH

Để dễ hình dung, một trình biên dịch được chia thành các giai đoạn, mỗi giai đoạn chuyển chương trình nguồn từ một dạng biểu diễn này sang một dạng biểu diễn khác. Một cách phân rã điển hình trình biên dịch được trình bày trong hình sau.



**Hình 1.6 - Các giai đoạn của một trình biên dịch**

Việc quản lý bảng ký hiệu và xử lý lỗi được thực hiện xuyên suốt qua tất cả các giai đoạn.

#### 1. Quản lý bảng ký hiệu

Một nhiệm vụ quan trọng của trình biên dịch là ghi lại các định danh được sử dụng trong chương trình nguồn và thu thập các thông tin về các thuộc tính khác nhau của mỗi định danh. Những thuộc tính này có thể cung cấp thông tin về vị trí lưu trữ được cấp phát cho một định danh, kiểu và tầm vực của định danh, và nếu định danh là tên của một thủ tục thì thuộc tính là các thông tin về số lượng và kiểu của các đối số, phương pháp truyền đối số và kiểu trả về của thủ tục nếu có.

Bảng ký hiệu (symbol table) là một cấu trúc dữ liệu mà mỗi phần tử là một mẫu tin dùng để lưu trữ một định danh, bao gồm các trường lưu giữ ký hiệu và các thuộc tính của nó. Cấu trúc này cho phép tìm kiếm, truy xuất danh biểu một cách nhanh chóng.

Trong quá trình phân tích từ vựng, danh biểu được tìm thấy và nó được đưa vào bảng ký hiệu nhưng nói chung các thuộc tính của nó có thể chưa xác định được trong giai đoạn này.

**Ví dụ 1.6:** Chẳng hạn, một khai báo trong Pascal có dạng

**var position, initial, rate : real**

thì thuộc tính kiểu real chưa thể xác định khi các danh biểu được xác định và đưa vào bảng ký hiệu. Các giai đoạn sau đó như phân tích ngữ nghĩa và sinh mã trung gian mới đưa thêm các thông tin này vào và sử dụng chúng. Nói chung giai đoạn sinh mã thường đưa các thông tin chi tiết về vị trí lưu trữ dành cho định danh và sẽ sử dụng chúng khi cần thiết.

Bảng ký hiệu

1	position	...
2	initial	...
3	rate	...
4		

## 2. Xử lý lỗi

Mỗi giai đoạn có thể gặp nhiều lỗi, tuy nhiên sau khi phát hiện ra lỗi, tùy thuộc vào trình biên dịch mà có các cách xử lý lỗi khác nhau, chẳng hạn :

- Dừng và thông báo lỗi khi gặp lỗi đầu tiên (Pascal).
- Ghi nhận lỗi và tiếp tục quá trình dịch (C).

Giai đoạn phân tích từ vựng thường gặp lỗi khi các ký tự không thể ghép thành một token.

Giai đoạn phân tích cú pháp gặp lỗi khi các token không thể kết hợp với nhau theo đúng cấu trúc ngôn ngữ.

Giai đoạn phân tích ngữ nghĩa báo lỗi khi các toán hạng có kiểu không đúng yêu cầu của phép toán hay các kết cấu không có nghĩa đối với thao tác thực hiện mặc dù chúng hoàn toàn đúng về mặt cú pháp.

## 3. Các giai đoạn phân tích

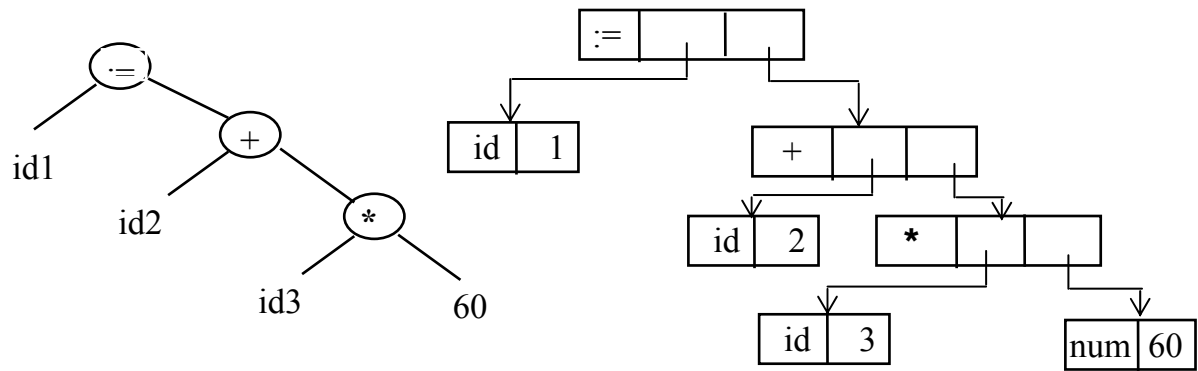
**Giai đoạn phân tích từ vựng:** Đọc từng ký tự gộp lại thành token, token có thể là một danh biểu, từ khóa, một ký hiệu,... Chuỗi ký tự tạo thành một token gọi là lexeme - trị từ vựng của token đó.

**Ví dụ 1.7:** Danh biểu rate có token id, trị từ vựng là rate và danh biểu này sẽ được đưa vào bảng ký hiệu nếu nó chưa có trong đó.

**Giai đoạn phân tích cú pháp và phân tích ngữ nghĩa:** Xây dựng cấu trúc phân cấp cho chuỗi các token, biểu diễn bởi cây cú pháp và kiểm tra ngôn ngữ theo cú pháp.

**Ví dụ 1.8:** Cây cú pháp và cấu trúc lưu trữ cho biểu thức

$$\text{position} := \text{initial} + \text{rate} * 60$$



**Hình 1.7 - Cây cú pháp và cấu trúc lưu trữ**

#### 4. Sinh mã trung gian

Sau khi phân tích ngữ nghĩa, một số trình biên dịch sẽ tạo ra một dạng biểu diễn trung gian của chương trình nguồn. Chúng ta có thể xem dạng biểu diễn này như một chương trình dành cho một máy trừu tượng. Chúng có 2 đặc tính quan trọng : dễ sinh và dễ dịch thành chương trình đích.

Dạng biểu diễn trung gian có rất nhiều loại. Thông thường, người ta sử dụng dạng "mã máy 3 địa chỉ" (three-address code), tương tự như dạng hợp ngữ cho một máy mà trong đó mỗi vị trí bộ nhớ có thể đóng vai trò như một thanh ghi.

Mã máy 3 địa chỉ là một dãy các lệnh liên tiếp, mỗi lệnh có thể có tối đa 3 đối số.

**Ví dụ 1.9:**

```

t1 := inttoreal (60)
t2 := id3 * t1
t3 := id2 + t2
id1 := t3

```

Dạng trung gian này có một số tính chất:

- Mỗi lệnh chỉ chứa nhiều nhất một toán tử. Do đó khi tạo ra lệnh này, trình biên dịch phải xác định thứ tự các phép toán, ví dụ \* thực hiện trước +.
- Trình biên dịch phải tạo ra một biến tạm để lưu trữ giá trị tính toán cho mỗi lệnh.
- Một số lệnh có ít hơn 3 toán hạng.

#### 5. Tối ưu mã

Giai đoạn tối ưu mã cố gắng cải thiện mã trung gian để có thể có mã máy thực hiện nhanh hơn. Một số phương pháp tối ưu hóa hoàn toàn bình thường.

**Ví dụ 1.10:**

Mã trung gian nêu trên có thể tối ưu thành:

```

t1 := id3 * 60.0
id1 := id2 + t1

```

Để tối ưu mã, ta thấy việc đổi số nguyên 60 thành số thực 60.0 có thể thực hiện một lần vào lúc biên dịch, vì vậy có thể loại bỏ phép toán inttoreal. Ngoài ra, t3 chỉ được dùng một lần để chuyển giá trị cho id1 nên có thể giảm bớt.



Có một khác biệt rất lớn giữa khối lượng tối ưu hoá mã được các trình biên dịch khác nhau thực hiện. Trong những trình biên dịch gọi là "trình biên dịch chuyên tối ưu", một phần thời gian đáng kể được dành cho giai đoạn này. Tuy nhiên, cũng có những phương pháp tối ưu giúp giảm đáng kể thời gian chạy của chương trình nguồn mà không làm chậm đi thời gian dịch quá nhiều.

## 6. Sinh mã

Giai đoạn cuối cùng của biên dịch là sinh mã đích, thường là mã máy hoặc mã hợp ngữ. Các vị trí vùng nhớ được chọn lựa cho mỗi biến được chương trình sử dụng. Sau đó, các chỉ thị trung gian được dịch lần lượt thành chuỗi các chỉ thị mã máy. Vấn đề quyết định là việc gán các biến cho các thanh ghi.

### Ví dụ 1.11:

Sử dụng các thanh ghi (chẳng hạn R1, R2) cho việc sinh mã đích như sau:

```
MOVF    id3, R2
MULF    #60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

Toán hạng thứ nhất và thứ hai của mỗi chỉ thị tương ứng mô tả đối tượng nguồn và đích. Chữ F trong mỗi chỉ thị cho biết chỉ thị đang xử lý các số chấm động (floating\_point). Dấu # để xác định số 60.0 xem như một hằng số.

## 7. Ví dụ

Xem hình vẽ 1.8 (trang 10) mô tả các giai đoạn biên dịch cho biểu thức:  
$$\text{position} := \text{initial} + \text{rate} * 60.$$

## IV. NHÓM CÁC GIAI ĐOẠN

Các giai đoạn mà chúng ta đề cập ở trên là thực hiện theo trình tự logic của một trình biên dịch. Nhưng trong thực tế, cài đặt các hoạt động của nhiều hơn một giai đoạn có thể được nhóm lại với nhau. Thông thường chúng được nhóm thành hai nhóm cơ bản, gọi là: kỳ đầu (Front end) và kỳ sau (Back end).

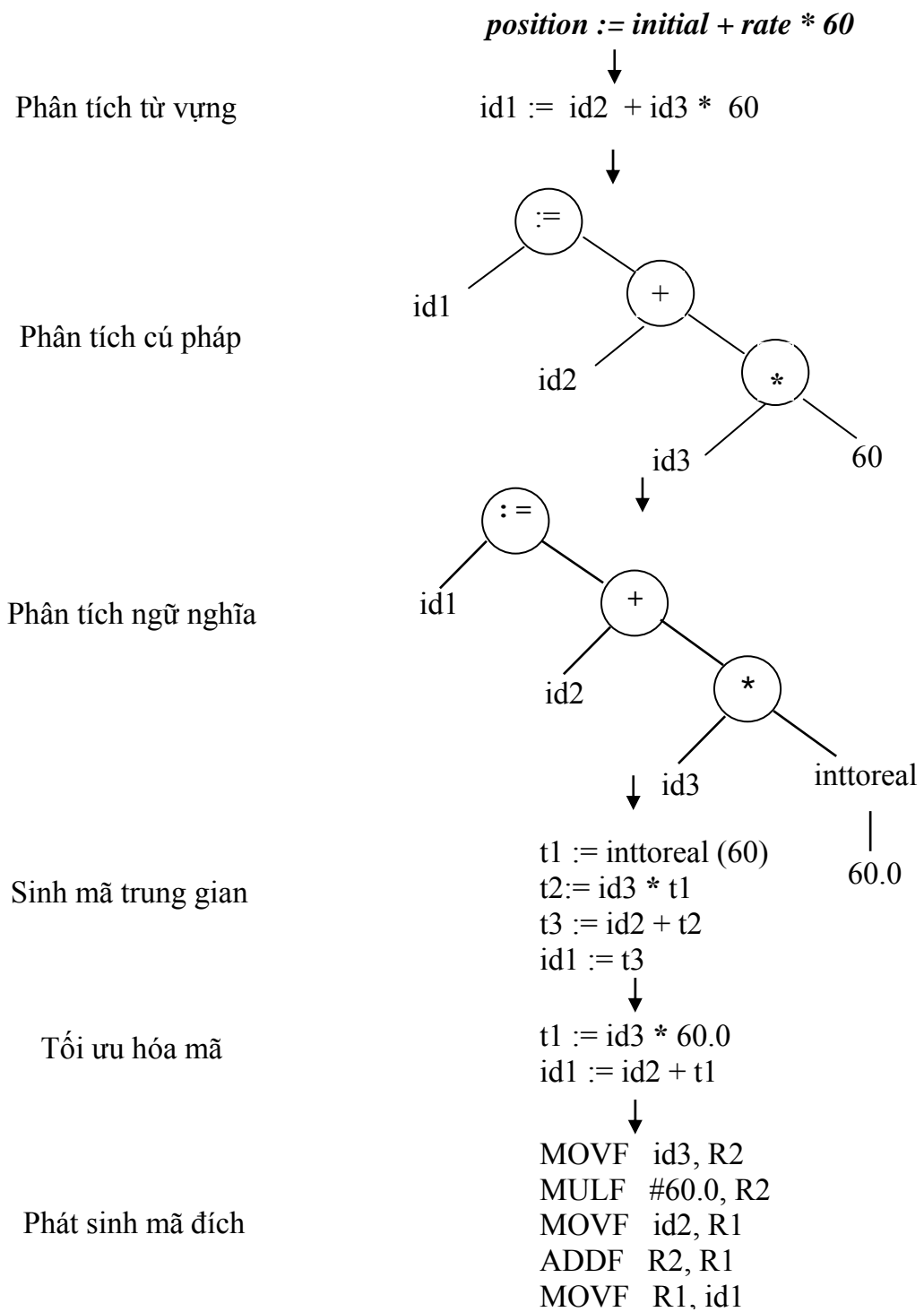
### 1. Kỳ đầu (Front End)

Kỳ đầu bao gồm các giai đoạn hoặc các phần giai đoạn phụ thuộc nhiều vào ngôn ngữ nguồn và hầu như độc lập với máy đích. Thông thường, nó chứa các giai đoạn sau: Phân tích từ vựng, Phân tích cú pháp, Phân tích ngữ nghĩa và Sinh mã trung gian. Một phần của công việc tối ưu hóa mã cũng được thực hiện ở kỳ đầu.

Front end cũng bao gồm cả việc xử lý lỗi xuất hiện trong từng giai đoạn.

### 2. Kỳ sau (Back End)

Kỳ sau bao gồm một số phần nào đó của trình biên dịch phụ thuộc vào máy đích và nói chung các phần này không phụ thuộc vào ngôn ngữ nguồn mà là ngôn ngữ trung gian. Trong kỳ sau, chúng ta gặp một số vấn đề tối ưu hoá mã, phát sinh mã đích cùng với việc xử lý lỗi và các thao tác trên bảng ký hiệu.



**Hình 1.8** - Minh họa các giai đoạn biên dịch một biểu thức

## CHƯƠNG II

### MỘT TRÌNH BIÊN DỊCH ĐƠN GIẢN

#### Nội dung chính:

Chương này giới thiệu một *trình biên dịch cho các biểu thức số học đơn giản* (trình biên dịch đơn giản) gồm hai kỳ: Kỳ đầu (Front end) và kỳ sau (Back end). Nội dung chính của chương tập trung vào *kỳ đầu* gồm các giai đoạn: Phân tích từ vựng, phân tích cú pháp và sinh mã trung gian với mục đích chuyển một biểu thức số học đơn giản từ dạng trung tố sang hậu tố. Kỳ sau chuyển đổi biểu thức ở dạng hậu tố sang *mã máy ảo kiểu stack*, sau đó sẽ thực thi đoạn mã đó trên *máy ảo kiểu stack* để cho ra kết quả tính toán cuối cùng.

#### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải nắm được:

- Các thành phần cấu tạo nên trình biên dịch đơn giản.
- Hoạt động và cách cài đặt các giai đoạn của kỳ trước của một trình biên dịch đơn giản.
- Cách sử dụng máy trừu tượng kiểu stack để chuyển đổi các biểu thức hậu tố sang mã máy ảo và cách thực thi các đoạn mã ảo này để có được kết quả cuối cùng.

#### Kiến thức cơ bản

Để tiếp nhận các nội dung được trình bày trong chương 2, sinh viên phải:

- Biết một ngôn ngữ lập trình nào đó: C, Pascal, v.v để hiểu cách cài đặt trình biên dịch.
- Có kiến thức về cấu trúc dữ liệu để hiểu cách tổ chức dữ liệu khi thực hiện cài đặt.

#### Tài liệu tham khảo:

[1] **Trình Biên Dịch** - Phan Thị Tươi (Trường Đại học kỹ thuật Tp.HCM) - NXB Giáo dục, 1998.

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

## I. ĐỊNH NGHĨA CÚ PHÁP

### 1. Văn phạm phi ngữ cảnh

Để xác định cú pháp của một ngôn ngữ, người ta dùng văn phạm phi ngữ cảnh CFG (Context Free Grammar) hay còn gọi là văn phạm BNF (Backers Naur Form)

Văn phạm phi ngữ cảnh bao gồm bốn thành phần:

1. Một tập hợp các token - các ký hiệu kết thúc (terminal symbols).

Ví dụ: Các từ khóa, các chuỗi, dấu ngoặc đơn, ...

2. Một tập hợp các ký hiệu chưa kết thúc (nonterminal symbols), còn gọi là các biến (variables).

Ví dụ: Câu lệnh, biểu thức, ...

3. Một tập hợp các luật sinh (productions) trong đó mỗi luật sinh bao gồm một ký hiệu chưa kết thúc - gọi là vế trái, một mũi tên và một chuỗi các token và / hoặc các ký hiệu chưa kết thúc gọi là vế phải.
4. Một trong các ký hiệu chưa kết thúc được dùng làm ký hiệu bắt đầu của văn phạm.

Chúng ta qui ước:

- Mô tả văn phạm bằng cách liệt kê các luật sinh.
- Luật sinh chứa ký hiệu bắt đầu sẽ được liệt kê đầu tiên.
- Nếu có nhiều luật sinh có cùng vế trái thì nhóm lại thành một luật sinh duy nhất, trong đó các vế phải cách nhau bởi ký hiệu “|” đọc là “hoặc”.

**Ví dụ 2.1:** Xem biểu thức là một danh sách của các số phân biệt nhau bởi dấu + và dấu -. Ta có, văn phạm với các luật sinh sau sẽ xác định cú pháp của biểu thức.

list $\rightarrow$ list + digit				
list $\rightarrow$ list - digit	$\Leftrightarrow$			list $\rightarrow$ list + digit   list - digit   digit
list $\rightarrow$ digit				digit $\rightarrow$ 0   1   2 ...   9
digit $\rightarrow$ 0   1   2   ...   9				

Như vậy văn phạm phi ngữ cảnh ở đây là:

- Tập hợp các ký hiệu kết thúc: 0, 1, 2, ..., 9, +, -
- Tập hợp các ký hiệu chưa kết thúc: list, digit.
- Các luật sinh đã nêu trên.
- Ký hiệu chưa kết thúc bắt đầu: list.

**Ví dụ 2.2:**

Từ ví dụ 2.1 ta thấy: 9 - 5 + 2 là một list vì:

9 là một list vì nó là một digit.

9 - 5 là một list vì 9 là một list và 5 là một digit.

9 - 5 + 2 là một list vì 9 - 5 là một list và 2 là một digit.

**Ví dụ 2.3:**

Một list là một chuỗi các lệnh, phân cách bởi dấu ; của khối begin - end trong Pascal. Một danh sách rỗng các lệnh có thể có giữa begin và end.

Chúng ta xây dựng văn phạm bởi các luật sinh sau:

```

block       $\rightarrow$  begin opt_stmts end
opt_stmts  $\rightarrow$  stmt_list |  $\epsilon$ 
stmt_list  $\rightarrow$  stmt_list ; stmt | stmt
  
```

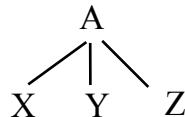
Trong đó `opt_stmts` (optional statements) là một danh sách các lệnh hoặc không có lệnh nào ( $\epsilon$ ).

Luật sinh cho `stmt_list` giống như luật sinh cho `list` trong ví dụ 2.1, bằng cách thay thế `+`, `-` bởi `;` và `stmt` thay cho `digit`.

## 2. Cây phân tích cú pháp (Parse Tree)

Cây phân tích cú pháp minh họa ký hiệu ban đầu của một văn phạm dẫn đến một chuỗi trong ngôn ngữ.

Nếu ký hiệu chưa kết thúc  $A$  có luật sinh  $A \rightarrow XYZ$  thì cây phân tích cú pháp có thể có một nút trong có nhãn  $A$  và có 3 nút con có nhãn tương ứng từ trái qua phải là  $X, Y, Z$ .



Một cách hình thức, cho một văn phạm phi ngữ cảnh thì cây phân tích cú pháp là một cây có các tính chất sau đây:

1. Nút gốc có nhãn là ký hiệu bắt đầu.
2. Mỗi một lá có nhãn là một ký hiệu kết thúc hoặc một  $\epsilon$ .
3. Mỗi nút trong có nhãn là một ký hiệu chưa kết thúc.
4. Nếu  $A$  là một ký hiệu chưa kết thúc được dùng làm nhãn cho một nút trong nào đó và  $X_1 \dots X_n$  là nhãn của các con của nó theo thứ tự từ trái sang phải thì  $A \rightarrow X_1 X_2 \dots X_n$  là một luật sinh. Ở đây  $X_1, \dots, X_n$  có thể là ký hiệu kết thúc hoặc chưa kết thúc. Đặc biệt, nếu  $A \rightarrow \epsilon$  thì nút có nhãn  $A$  có thể có một con có nhãn  $\epsilon$ .

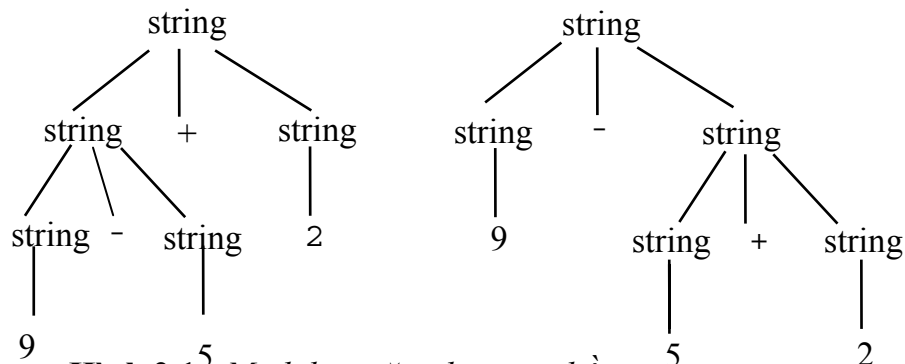
## 3. Sự mơ hồ của văn phạm

Một văn phạm có thể sinh ra nhiều hơn một cây phân tích cú pháp cho cùng một chuỗi nhập thì gọi là văn phạm mơ hồ.

**Ví dụ 2.4:** Giả sử chúng ta không phân biệt một list với một digit, xem chúng đều là một string ta có văn phạm:

$\text{string} \rightarrow \text{string} + \text{string} \mid \text{string} - \text{string} \mid 0 \mid 1 \mid \dots \mid 9.$

Với văn phạm này thì chuỗi biểu thức  $9 - 5 + 2$  có đến hai cây phân tích cú pháp như sau :



**Hình 2.1** - Minh họa văn phạm mơ hồ

Tương tự với cách đặt dấu ngoặc vào biểu thức như sau :

$$(9 - 5) + 2$$

$$9 - (5 + 2)$$

Bởi vì một chuỗi với nhiều cây phân tích cú pháp thường sẽ có nhiều nghĩa, do đó khi biên dịch các chương trình ứng dụng, chúng ta cần thiết kế các văn phạm không có sự mơ hồ hoặc cần bổ sung thêm các qui tắc cần thiết để giải quyết sự mơ hồ cho văn phạm.

#### 4. Sự kết hợp của các toán tử

Thông thường, theo quy ước ta có biểu thức  $9 + 5 + 2$  tương đương  $(9 + 5) + 2$  và  $9 - 5 - 2$  tương đương với  $(9 - 5) - 2$ . Khi một toán hạng như 5 có hai toán tử ở trái và phải thì nó phải chọn một trong hai để xử lý trước. Nếu toán tử bên trái được thực hiện trước ta gọi là **kết hợp trái**. Ngược lại là **kết hợp phải**.

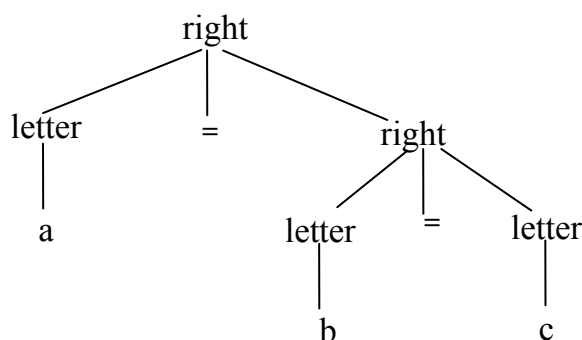
Thường thì bốn phép toán số học:  $+$ ,  $-$ ,  $*$ ,  $/$  có tính kết hợp trái. Các phép toán như số mũ, phép gán bằng ( $=$ ) có tính kết hợp phải.

**Ví dụ 2.5:** Trong ngôn ngữ C, biểu thức  $a = b = c$  tương đương  $a = (b = c)$  vì chuỗi  $a = b = c$  với toán tử kết hợp phải được sinh ra bởi văn phạm:

right  $\rightarrow$  letter = right | letter

letter  $\rightarrow$  a | b | ... | z

Ta có cây phân tích cú pháp có dạng như sau (chú ý hướng của cây nghiêng về bên phải trong khi cây cho các phép toán có kết hợp trái thường nghiêng về trái)



**Hình 2.2** - Minh họa cây phân tích cú pháp cho toán tử kết hợp phải

#### 5. Thứ tự ưu tiên của các toán tử

Xét biểu thức  $9 + 5 * 2$ . Có 2 cách để diễn giải biểu thức này, đó là  $9 + (5 * 2)$  hoặc  $(9 + 5) * 2$ . Tính kết hợp của phép  $+$  và  $*$  không giải quyết được sự mơ hồ này, vì vậy cần phải quy định một thứ tự ưu tiên giữa các loại toán tử khác nhau.

Thông thường trong toán học, các toán tử  $*$  và  $/$  có độ ưu tiên cao hơn  $+$  và  $-$ .

#### Cú pháp cho biểu thức :

Văn phạm cho các biểu thức số học có thể xây dựng từ bảng kết hợp và ưu tiên của các toán tử. Chúng ta có thể bắt đầu với bốn phép tính số học theo thứ bậc sau :

Kết hợp trái $+$ , $-$	↓	Thứ tự ưu tiên
Kết hợp trái $*$ , $/$		từ thấp đến cao

Chúng ta tạo hai ký hiệu chưa kết thúc  $\text{expr}$  và  $\text{term}$  cho hai mức ưu tiên và một ký hiệu chưa kết thúc  $\text{factor}$  làm đơn vị phát sinh cơ sở của biểu thức. Ta có đơn vị cơ bản trong biểu thức là số hoặc biểu thức trong dấu ngoặc.

$$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$$

Phép nhân và chia có thứ tự ưu tiên cao hơn đồng thời chúng kết hợp trái nên luật sinh cho  $\text{term}$  tương tự như cho  $\text{list}$  :

$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$

Tương tự, ta có luật sinh cho  $\text{expr}$  :

$$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$$

Vậy, cuối cùng ta thu được văn phạm cho biểu thức như sau :

$$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$$
$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$
$$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$$

Như vậy: Văn phạm này xem biểu thức như là một danh sách các  $\text{term}$  được phân cách nhau bởi dấu  $+$  hoặc  $-$ .  $\text{Term}$  là một list các  $\text{factor}$  phân cách nhau bởi  $*$  hoặc  $/$ . Chú ý rằng bất kỳ một biểu thức nào trong ngoặc đều là  $\text{factor}$ , vì thế với các dấu ngoặc chúng ta có thể xây dựng các biểu thức lồng sâu nhiều cấp tùy ý.

### Cú pháp các câu lệnh:

Từ khóa (keyword) cho phép chúng ta nhận ra câu lệnh trong hầu hết các ngôn ngữ. Ví dụ trong Pascal, hầu hết các lệnh đều bắt đầu bởi một từ khóa ngoại trừ lệnh gán. Một số lệnh Pascal được định nghĩa bởi văn phạm (mơ hồ) sau, trong đó  $\text{id}$  chỉ một danh biểu (tên biến).

$$\begin{aligned} \text{stmt} \rightarrow & \text{id} := \text{expr} \\ & \mid \text{if expr then stmt} \\ & \mid \text{if expr then stmt else stmt} \\ & \mid \text{while expr do stmt} \\ & \mid \text{begin opt\_stmts end} \end{aligned}$$

Ký hiệu chưa kết thúc  $\text{opt\_stmts}$  sinh ra một danh sách có thể rỗng các lệnh, phân cách nhau bởi dấu chấm phẩy (;)

## II. DỊCH TRỰC TIẾP CÚ PHÁP (Syntax - Directed Translation)

Để dịch một kết cấu ngôn ngữ lập trình, trong quá trình dịch, bộ biên dịch cần lưu lại nhiều đại lượng khác cho việc sinh mã ngoài mã lệnh cần tạo ra cho kết cấu. Chẳng hạn nó cần biết kiểu (type) của kết cấu, địa chỉ của lệnh đầu tiên trong mã đích, số lệnh phát sinh, v.v Vì vậy ta nói một cách ảo về thuộc tính (attribute) đi kèm theo kết cấu. Một thuộc tính có thể biểu diễn cho một đại lượng bất kỳ như một kiểu, một chuỗi, một địa chỉ vùng nhớ, v.v

Chúng ta sử dụng **định nghĩa trực tiếp cú pháp (syntax - directed definition)** nhằm đặc tả việc biên dịch các kết cấu ngôn ngữ lập trình theo các thuộc tính đi kèm

với thành phần cú pháp của nó. Chúng ta cũng sẽ sử dụng một thuật ngữ có tính thủ tục hơn là **lược đồ dịch (translation scheme)** để đặc tả quá trình dịch. Trong chương này, ta sử dụng lược đồ dịch để dịch một biểu thức trung tố thành dạng hậu tố.

## 1. Ký pháp hậu tố (Postfix Notation)

Ký pháp hậu tố của biểu thức E có thể được định nghĩa quy nạp như sau:

1. Nếu E là một biến hay hằng thì ký pháp hậu tố của E chính là E.
2. Nếu E là một biểu thức có dạng **E1 op E2** trong đó op là một toán tử hai ngôi thì ký pháp hậu tố của E là **E1' E2' op**. Trong đó E1', E2' tương ứng là ký pháp hậu tố của E1, E2.
3. Nếu E là một biểu thức dạng **(E1)** thì ký pháp hậu tố của E là ký pháp hậu tố của E1.

Trong dạng ký pháp hậu tố, dấu ngoặc là không cần thiết vì vị trí và số lượng các đối số chỉ cho phép xác định một sự giải mã duy nhất cho một biểu thức hậu tố.

**Ví dụ 2.6:** Dạng hậu tố của biểu thức  $(9 - 5) + 2$  là 9 5 - 2 +

Dạng hậu tố của biểu thức  $9 - (5 + 2)$  là 9 5 2 + -

## 2. Định nghĩa trực tiếp cú pháp (Syntax - Directed Definition)

Định nghĩa trực tiếp cú pháp sử dụng văn phạm phi ngữ cảnh để đặc tả cấu trúc cú pháp của dòng input nhập. Nó liên kết mỗi ký hiệu văn phạm với một tập các thuộc tính và mỗi luật sinh kết hợp với một tập các quy tắc ngữ nghĩa (semantic rule) để tính giá trị của thuộc tính đi kèm với những ký hiệu có trong luật sinh văn phạm. Văn phạm và tập các quy tắc ngữ nghĩa tạo nên định nghĩa trực tiếp cú pháp.

Phiên dịch (translation) là một ánh xạ giữa input - output (input - output mapping). Output cho mỗi input x được xác định theo cách sau. Trước hết xây dựng cây phân tích cú pháp cho x. Giả sử nút n trong cây phân tích cú pháp có nhãn là ký hiệu văn phạm X. Ta viết X.a để chỉ giá trị của thuộc tính a của X tại nút đó. Giá trị của X.a tại n được tính bằng cách sử dụng quy tắc ngữ nghĩa cho thuộc tính a kết hợp với luật sinh cho X tại nút n. Cây phân tích cú pháp có thể hiện rõ giá trị của thuộc tính tại mỗi nút gọi là cây phân tích cú pháp chú thích (annotated parse tree).

## 3. Thuộc tính tổng hợp (Synthesized Attributes)

Một thuộc tính được gọi là tổng hợp nếu giá trị của nó tại một nút trên cây cú pháp được xác định từ các giá trị của các thuộc tính tại các nút con của nút đó.

**Ví dụ 2.7:** Định nghĩa trực tiếp cú pháp cho việc dịch các biểu thức các số cách nhau bởi dấu + hoặc - thành ký pháp hậu tố như sau:

Luật sinh	Quy tắc ngữ nghĩa
$E \rightarrow E1 + T$	$E.t := E1.t \parallel T.t \parallel '+'$
$E \rightarrow E1 - T$	$E.t := E1.t \parallel T.t \parallel '-'$
$E \rightarrow T$	$E.t := T.t$
$T \rightarrow 0$	$T.t := '0'$
...	...



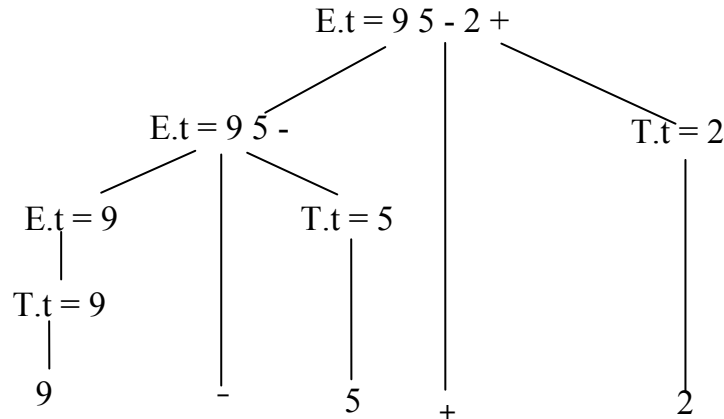
$T \rightarrow 9$

$T.t := '9'$

**Hình 2.3** - Ví dụ về định nghĩa trực tiếp cú pháp

Chẳng hạn, một quy tắc ngữ nghĩa  $E.t := E1.t \parallel T.t \parallel '+'$  kết hợp với luật sinh xác định giá trị của thuộc tính  $E.t$  bằng cách ghép các ký pháp hậu tố của  $E1.t$  và  $T.t$  và dấu '+'. Dấu  $\parallel$  có nghĩa như sự ghép các chuỗi.

Ta có cây phân tích cú pháp chủ thích cho biểu thức  $9 - 5 + 2$  như sau :



**Hình 2.4** - Minh họa cây phân tích cú pháp chủ thích

Giá trị của thuộc tính  $t$  tại mỗi nút được tính bằng cách dùng quy tắc ngữ nghĩa kết hợp với luật sinh tại nút đó. Giá trị thuộc tính tại nút gốc là ký pháp hậu tố của chuỗi được sinh ra bởi cây phân tích cú pháp.

#### 4. Duyệt theo chiều sâu (Depth - First Traversal)

Quá trình dịch được cài đặt bằng cách đánh giá các luật ngữ nghĩa cho các thuộc tính trong cây phân tích cú pháp theo một thứ tự xác định trước. Ta dùng phép duyệt cây theo chiều sâu để đánh giá quy tắc ngữ nghĩa. Bắt đầu từ nút gốc, thăm lần lượt (đệ qui) các con của mỗi nút theo thứ tự từ trái sang phải.

**Procedure** visit ( $n$  : node);

**begin**

for với mỗi nút con  $m$  của  $n$ , từ trái sang phải do

visit ( $m$ );

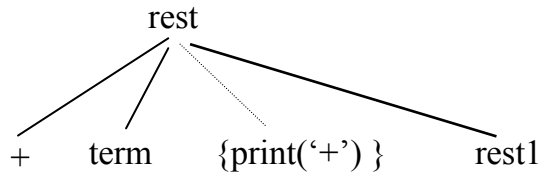
Đánh giá quy tắc ngữ nghĩa tại nút  $n$ ;

**end**

#### 5. Lược đồ dịch (Translation Scheme)

Một lược đồ dịch là một văn phạm phi ngữ cảnh, trong đó các đoạn chương trình gọi là hành vi ngữ nghĩa (semantic actions) được gán vào vế phải của luật sinh. Lược đồ dịch cũng như định nghĩa trực tiếp cú pháp nhưng thứ tự đánh giá các quy tắc ngữ nghĩa được trình bày một cách rõ ràng. Vị trí mà tại đó một hành vi được thực hiện được trình bày trong cặp dấu ngoặc nhọn  $\{ \}$  và viết vào vế phải luật sinh.

**Ví dụ 2.8:**  $rest \rightarrow + term \{print ('+')\} rest1$ .



**Hình 2.5** - Một nút lá được xây dựng cho hành vi ngữ nghĩa

Lược đồ dịch tạo ra một output cho mỗi câu nhập x sinh ra từ văn phạm đã cho bằng cách thực hiện các hành vi theo thứ tự mà chúng xuất hiện trong quá trình duyệt theo chiều sâu cây phân tích cú pháp của x. Chẳng hạn, xét cây phân tích cú pháp với một nút có nhãn rest biểu diễn luật sinh nói trên. Hành vi ngữ nghĩa { print( '+' ) } được thực hiện sau khi cây con term được duyệt nhưng trước khi cây con rest1 được thăm.

## 6. Phát sinh bản dịch (Emitting a Translation)

Trong chương này, hành vi ngữ nghĩa trong lược đồ dịch sẽ ghi kết quả của quá trình phiên dịch vào một tập tin, mỗi lần một chuỗi hoặc một ký tự. Chẳng hạn, khi dịch  $9 - 5 + 2$  thành  $9\ 5 - 2 +$  bằng cách ghi mỗi ký tự trong  $9 - 5 + 2$  đúng một lần mà không phải ghi lại quá trình dịch của các biểu thức con. Khi tạo ra output dần dần theo cách này, thứ tự in ra các ký tự sẽ rất quan trọng.

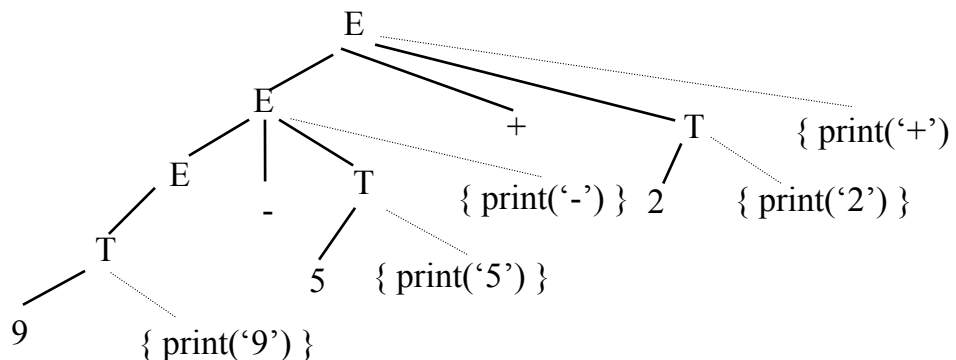
Chú ý rằng các định nghĩa trực tiếp cú pháp đều có đặc điểm sau: chuỗi biểu diễn cho bản dịch của ký hiệu chưa kết thúc ở vế trái của mỗi luật sinh là sự ghép nối của các bản dịch ở vế phải theo đúng thứ tự của chúng trong luật sinh và có thể thêm một số chuỗi khác xen vào giữa. Một định nghĩa trực tiếp cú pháp theo dạng này được xem là đơn giản.

**Ví dụ 2.9:** Với định nghĩa trực tiếp cú pháp như hình 2.3, ta xây dựng lược đồ dịch như sau :

$E \rightarrow E1 + T$	{ print ( '+' ) }
$E \rightarrow E1 - T$	{ print ( '-' ) }
$E \rightarrow T$	
$T \rightarrow 0$	{ print ( '0' ) }
....	
$T \rightarrow 9$	{ print ( '9' ) }

**Hình 2.6** - Lược đồ dịch biểu thức trung tố thành hậu tố

Ta có các hành động dịch biểu thức  $9 - 5 + 2$  thành  $9\ 5 - 2 +$  như sau :



**Hình 2.7** - Các hành động dịch biểu thức  $9-5+2$  thành  $9\ 5- 2 +$

Xem như một quy tắc tổng quát, phần lớn các phương pháp phân tích cú pháp đều xử lý input của chúng từ trái sang phải, trong lược đồ dịch đơn giản (lược đồ dịch dẫn xuất từ một định nghĩa trực tiếp cú pháp đơn giản), các hành vi ngữ nghĩa cũng được thực hiện từ trái sang phải. Vì thế, để cài đặt một lược đồ dịch đơn giản, chúng ta có thể thực hiện các hành vi ngữ nghĩa trong lúc phân tích cú pháp mà không nhất thiết phải xây dựng cây phân tích cú pháp.

### III. PHÂN TÍCH CÚ PHÁP (PARSING)

Phân tích cú pháp là quá trình xác định xem liệu một chuỗi ký hiệu kết thúc (token) có thể được sinh ra từ một văn phạm hay không? Khi nói về vấn đề này, chúng ta xem như đang xây dựng một cây phân tích cú pháp, mặc dù một trình biên dịch có thể không xây dựng một cây như thế. Tuy nhiên, quá trình phân tích cú pháp (parse) phải có khả năng xây dựng nó, nếu không thì việc biên dịch sẽ không bảo đảm được tính đúng đắn.

Phần lớn các phương pháp phân tích cú pháp đều rơi vào một trong 2 lớp: phương pháp phân tích từ trên xuống và phương pháp phân tích từ dưới lên. Những thuật ngữ này muốn đề cập đến thứ tự xây dựng các nút trong cây phân tích cú pháp. Trong phương pháp đầu, quá trình xây dựng bắt đầu từ gốc tiến hành hướng xuống các nút lá, còn trong phương pháp sau thì thực hiện từ các nút lá hướng về gốc. Phương pháp phân tích từ trên xuống thông dụng hơn nhờ vào tính hiệu quả của nó khi xây dựng theo lối thủ công. Ngược lại, phương pháp phân tích từ dưới lên lại có thể xử lý được một lớp văn phạm và lược đồ dịch phong phú hơn. Vì vậy, đa số các công cụ phần mềm giúp xây dựng thể phân tích cú pháp một cách trực tiếp từ văn phạm đều có xu hướng sử dụng phương pháp từ dưới lên.

#### 1. Phân tích cú pháp từ trên xuống (Top - Down Parsing)

Xét văn phạm sinh ra một tập con các kiểu dữ liệu của Pascal

`type → simple | ↑ id | array [simple] of type`

`simple → integer | char | num .. num`

Phân tích trên xuống bắt đầu bởi nút gốc, nhận là ký hiệu chưa kết thúc bắt đầu và lặp lại việc thực hiện hai bước sau đây:

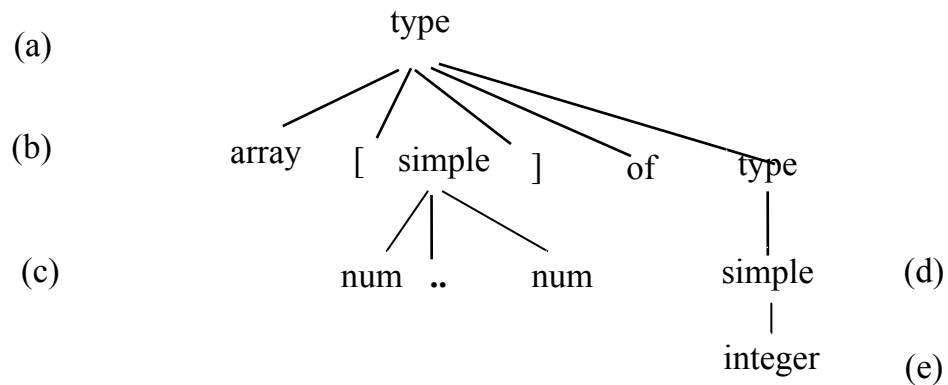
1. Tại nút  $n$ , nhận là ký hiệu chưa kết thúc  $A$ , chọn một trong những luật sinh của  $A$  và xây dựng các con của  $n$  cho các ký hiệu trong vế phải của luật sinh.
2. Tìm nút kế tiếp mà tại đó một cây con sẽ được xây dựng. Đối với một số văn phạm, các bước trên được cài đặt bằng một phép quét (scan) dòng nhập từ trái qua phải.

**Ví dụ 2.10:** Với các luật sinh của văn phạm trên, ta xây dựng cây cú pháp cho dòng nhập: `array [num .. num] of integer`

Mở đầu ta xây dựng nút gốc với nhãn **type**. Để xây dựng các nút con của **type** ta chọn luật sinh `type → array [simple] of type`. Các ký hiệu nằm bên phải của luật sinh này là **array**, **[**, **simple**, **]**, **of**, **type** do đó nút gốc **type** có 6 con có nhãn tương ứng (áp dụng bước 1)

Trong các nút con của **type**, từ trái qua thì nút con có nhãn **simple** (một ký hiệu chưa kết thúc) do đó có thể xây dựng một cây con tại nút **simple** (bước 2)

Trong các luật sinh có vẻ trái là simple, ta chọn luật sinh **simple**  $\rightarrow$  **num .. num** để xây dựng. Nói chung, việc chọn một luật sinh có thể được xem như một quá trình **thử và sai** (trial - and - error). Nghĩa là một luật sinh được chọn để thử và sau đó quay lại để thử một luật sinh khác nếu luật sinh ban đầu không phù hợp. Một luật sinh là không phù hợp nếu sau khi sử dụng luật sinh này chúng ta không thể xây dựng một cây hợp với dòng nhập. Để tránh việc lẩn ngược, người ta đưa ra một phương pháp gọi là phương pháp phân tích cú pháp dự đoán.



**Hình 2.8** - Minh họa quá trình phân tích cú pháp từ trên xuống

## 2. Phân tích cú pháp dự đoán (Predictive Parsing)

Phương pháp phân tích cú pháp đệ qui xuống (recursive-descent parsing) là một phương pháp phân tích từ trên xuống, trong đó chúng ta thực hiện một loạt thủ tục đệ qui để xử lý chuỗi nhập. Mỗi một thủ tục kết hợp với một ký hiệu chưa kết thúc của văn phạm. Ở đây chúng ta xét một trường hợp đặc biệt của phương pháp đệ qui xuống là phương pháp phân tích dự đoán trong đó ký hiệu dò tìm sẽ xác định thủ tục được chọn đối với ký hiệu chưa kết thúc. Chuỗi các thủ tục được gọi trong quá trình xử lý chuỗi nhập sẽ tạo ra cây phân tích cú pháp.

**Ví dụ 2.11:** Xét văn phạm như trên, ta viết các thủ tục type và simple tương ứng với các ký hiệu chưa kết thúc type và simple trong văn phạm. Ta còn đưa thêm thủ tục match để đơn giản hóa đoạn mã cho hai thủ tục trên, nó sẽ dịch tới ký hiệu kế tiếp nếu tham số t của nó so khớp với ký hiệu dò tìm tại đầu đọc (lookahead).

```

procedure match (t: token);
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end;

procedure type;
begin
    if lookahead in [integer, char, num] then
        simple
    else if lookahead = '↑' then begin

```

```

        match ('↑'); match(id);
    end
    else if lookahead = array then begin
        match(array); match(['');
        simple;
        match(']'); match(of);
        type
    end
    else error;
end;
procedure simple;
begin
    if lookahead = integer then match(integer)
    else if lookahead = char then match(char)
    else if lookahead = num then
        begin
            match(num); match(dotdot); match(num);
        end
    else error
    end;
end;

```

**Hình 2.9** - Đoạn mã giả minh họa phương pháp phân tích dự đoán

Phân tích cú pháp bắt đầu bằng lời gọi tới thủ tục cho ký hiệu bắt đầu `type`. Với dòng nhập `array [num .. num] of integer` thì đầu đọc lookahead bắt đầu sẽ đọc token `array`. Thủ tục **type** sau đó sẽ thực hiện chuỗi lệnh: `match(array); match([''); simple; match(']'); match(of); type`. Sau khi đã đọc được **array** và `[` thì ký hiệu hiện tại là **num**. Tại điểm này thì thủ tục **simple** và các lệnh `match(num); match(dotdot); match(num)` được thực hiện.

Xét luật sinh **type**  $\rightarrow$  **simple**. Luật sinh này có thể được dùng khi ký hiệu dò tìm sinh ra bởi **simple**, chẳng hạn ký hiệu dò tìm là *integer* mặc dù trong văn phạm không có luật sinh **type**  $\rightarrow$  **integer**, nhưng có luật sinh **simple**  $\rightarrow$  **integer**, do đó luật sinh **type**  $\rightarrow$  **simple** được dùng bằng cách trong **type** gọi **simple**.

Phân tích dự đoán dựa vào thông tin về các ký hiệu đầu sinh ra bởi vế phải của một luật sinh. Nói chính xác hơn, giả sử ta có luật sinh  $A \rightarrow \gamma$ , ta định nghĩa tập hợp :

$FIRST(\gamma) = \{ \text{token} \mid \text{xuất hiện như các ký hiệu đầu của một hoặc nhiều chuỗi sinh ra bởi } \gamma \}$ . Nếu  $\gamma$  là  $\epsilon$  hoặc có thể sinh ra  $\epsilon$  thì  $\epsilon \in FIRST(\gamma)$ .

**Ví dụ 2.12:** Xét văn phạm như trên, ta dễ dàng xác định:

$FIRST(\text{ simple}) = \{ \text{integer, char, num} \}$

$$\text{FIRST}(\uparrow \text{id}) = \{ \uparrow \}$$

$$\text{FIRST}(\text{array} [\text{simple}] \text{ of type}) = \{ \text{array} \}$$

Nếu ta có  $A \rightarrow \alpha$  và  $A \rightarrow \beta$ , phân tích đệ qui xuống sẽ không phải quay lui nếu  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ . Nếu ký hiệu dò tìm thuộc  $\text{FIRST}(\alpha)$  thì  $A \rightarrow \alpha$  được dùng. Ngược lại, nếu ký hiệu dò tìm thuộc  $\text{FIRST}(\beta)$  thì  $A \rightarrow \beta$  được dùng.

### Trường hợp $\alpha = \varepsilon$ (Luật sinh $\varepsilon$ )

**Ví dụ 2.13:** Xét văn phạm chứa các luật sinh sau :

$$\text{stmt} \rightarrow \text{begin opt\_stmts end}$$

$$\text{opt\_stmts} \rightarrow \text{stmt\_list} \mid \varepsilon$$

Khi phân tích cú pháp cho  $\text{opt\_stmts}$ , nếu ký hiệu dò tìm  $\notin \text{FIRST}(\text{stmt\_list})$  thì sử dụng luật sinh:  **$\text{opt\_stmts} \rightarrow \varepsilon$** . Chọn lựa này hoàn toàn chính xác nếu ký hiệu dò tìm là **end**, mọi ký hiệu dò tìm khác **end** sẽ gây ra lỗi và được phát hiện trong khi phân tích  $\text{stmt}$ .

### 3. Thiết kế bộ phân tích cú pháp dự đoán

Bộ phân tích dự đoán là một chương trình bao gồm các thủ tục tương ứng với các ký hiệu chưa kết thúc. Mỗi thủ tục sẽ thực hiện hai công việc sau:

1. Luật sinh mà về phải  $\alpha$  của nó sẽ được dùng nếu ký hiệu dò tìm thuộc  $\text{FIRST}(\alpha)$ . Nếu có một sự đụng độ giữa hai về phải đối với bất kỳ một ký hiệu dò tìm nào thì không thể dùng phương pháp này. Một luật sinh với  $\varepsilon$  nằm bên về phải được dùng nếu ký hiệu dò tìm không thuộc tập hợp  $\text{FIRST}$  của bất kỳ về phải nào khác.

2. Một ký hiệu chưa kết thúc tương ứng lời gọi thủ tục, một token phải phù hợp với ký hiệu dò tìm. Nếu token không phù hợp với ký hiệu dò tìm thì có lỗi.

### 4. Loại bỏ đệ qui trái

Một bộ phân tích cú pháp đệ quy xuống có thể sẽ dẫn đến một vòng lặp vô tận nếu gặp một luật sinh đệ qui trái dạng  $E \rightarrow E + T$  bởi vì ký hiệu trái nhất bên về phải cũng giống như ký hiệu chưa kết thúc bên về trái của luật sinh.

Để giải quyết được vấn đề này chúng ta phải loại bỏ đệ qui trái bằng cách thêm vào một ký hiệu chưa kết thúc mới. Chẳng hạn với luật sinh dạng  $A \rightarrow A\alpha \mid \beta$ . Ta thêm vào một ký hiệu chưa kết thúc  $R$  để viết lại thành tập luật sinh như sau :

$$A \rightarrow \beta R$$

$$R \rightarrow \alpha R \mid \varepsilon$$

**Ví dụ 2.14:** Xét luật sinh đệ qui trái :  $E \rightarrow E + T \mid T$

Sử dụng quy tắc khử đệ qui trái nói trên với :  $A \cong E, \alpha \cong + T, \beta \cong T$ .

Luật sinh trên có thể biến đổi tương đương thành tập luật sinh :

$$E \rightarrow T R$$

$$R \rightarrow + T R \mid \varepsilon$$

#### IV. MỘT CHƯƠNG TRÌNH DỊCH BIỂU THỨC ĐƠN GIẢN

Sử dụng các kỹ thuật nêu trên, chúng ta xây dựng một bộ dịch trực tiếp cú pháp mà nó dịch một biểu thức số học đơn giản từ trung tố sang hậu tố. Ta bắt đầu với các biểu thức là các chữ số viết cách nhau bởi + hoặc -.

Xét lược đồ dịch cho dạng biểu thức này :

$\text{expr} \rightarrow \text{expr} + \text{term} \quad \{ \text{print}(' + ') \}$

$\text{expr} \rightarrow \text{expr} - \text{term} \quad \{ \text{print}(' - ') \}$

$\text{expr} \rightarrow \text{term}$

$\text{term} \rightarrow 0 \quad \{ \text{print}('0') \}$

...

$\text{term} \rightarrow 9 \quad \{ \text{print}('9') \}$

**Hình 2.10** - Đặc tả lược đồ dịch khởi đầu

Văn phạm nền tảng cho lược đồ dịch trên có chứa luật sinh đệ quy trái, bộ phân tích cú pháp dự đoán không xử lý được văn phạm dạng này, cho nên ta cần loại bỏ đệ quy trái bằng cách đưa vào một ký hiệu chưa kết thúc mới rest để được văn phạm thích hợp như sau:

$\text{expr} \rightarrow \text{term rest}$

$\text{rest} \rightarrow + \text{term} \{ \text{print}(' + ') \} \text{rest} \mid - \text{term} \{ \text{print}(' - ') \} \text{rest} \mid \epsilon$

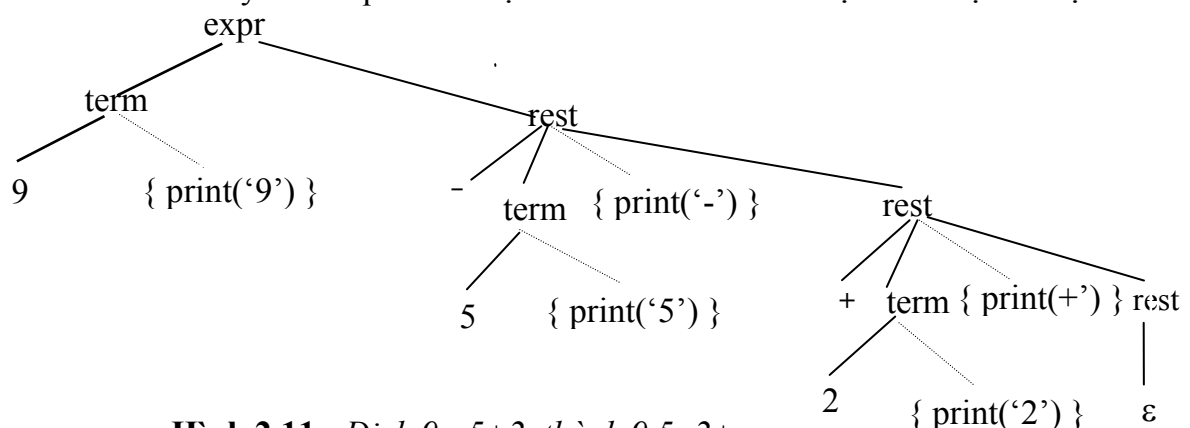
$\text{term} \rightarrow 0 \{ \text{print}('0') \}$

$\text{term} \rightarrow 1 \{ \text{print}('1') \}$

...

$\text{term} \rightarrow 9 \{ \text{print}('9') \}$

Hình sau đây mô tả quá trình dịch biểu thức  $9 - 5 + 2$  dựa vào lược đồ dịch trên:



**Hình 2.11** - Dịch  $9 - 5 + 2$  thành  $9\ 5 - 2 +$

Bây giờ ta cài đặt chương trình dịch bằng C theo đặc tả như trên. Phần chính của chương trình này là các đoạn mã C cho các hàm `expr`, `term` và `rest`.

// Hàm `expr( )` tương ứng với ký hiệu chưa kết thúc `expr`

**expr( )**

```

{
    term( ) ; rest( );
}

```

// Hàm expr( ) tương ứng với ký hiệu chưa kết thúc expr  
**rest( )**

```

{
    if (lookahead == '+' ) {
        match('+') ; term( ) ; putchar ('+' ) ; rest( );
    }
    else if (lookahead == '-') {
        match('-') ; term( ) ; putchar ('-' ) ; rest( );
    }
    else ;
}

```

// Hàm expr( ) tương ứng với ký hiệu chưa kết thúc expr  
**term( )**

```

{
    if (isdigit (lookahead) ) {
        putchar (lookahead); match (lookahead);
    }
    else error( );
}

```

### **Tối ưu hóa chương trình dịch**

Một số lời gọi đệ quy có thể được thay thế bằng các vòng lặp để làm cho chương trình thực hiện nhanh hơn. Đoạn mã cho rest có thể được viết lại như sau :

```

rest( )
{
    L : if (lookahead == '+' ) {
        match('+') ; term( ) ; putchar ('+' ) ; goto L;
    }
    else if (lookahead == '-') {
        match('-') ; term( ) ; putchar ('-' ) ; goto L;
    }
}

```



```

    }
    else ;
}

```

Nhờ sự thay thế này, hai hàm `rest` và `expr` có thể được tích hợp lại thành một.

Mặt khác, trong C, một câu lệnh `stmt` có thể được thực hiện lặp đi lặp lại bằng cách viết : `while (1) stmt` với 1 là điều kiện hằng đúng. Chúng ta cũng có thể thoát khỏi vòng lặp dễ dàng bằng lệnh `break`.

Đoạn chương trình có thể được viết lại như sau :

```

expr ( )
{
    term ( )
    while (1)
        if (lookahead == '+' ) {
            match('+') ; term( ) ; putchar ('+' ) ;
        }
        else if (lookahead == '-') {
            match('-') ; term( ) ; putchar ('-' ) ;
        }
        else break;
}

```

### Chương trình C dịch biểu thức trung tố sang hậu tố

Chương trình nguồn C hoàn chỉnh cho chương trình dịch có mã như sau :

```

#include< ctype.h>                /* nạp tập tin chứa isdigit vào*/
int lookahead;

main ( )
{
    lookahead = getchar( );
    expr( ) ; putchar('\n');      /* thêm vào ký tự xuống hàng */
}

expr( )
{
    term( );
    while(1)

```

```

    if (lookahead == '+')
        { match('+'); term( ); putchar('+ '); }
    else if (lookahead == '-')
        { match('-'); term( ); putchar('- '); }
    else break;
}

term( )
{
    if (isdigit(lookahead))
        { putchar(lookahead); match(lookahead); }
    else error( );
}

match ( int t)
{
    if (lookahead == t)
        lookahead = getchar();
    else error( );
}

error( )
{
    printf("syntax error \n");    /* in ra thông báo lỗi */
    exit(1);                      /* rồi kết thúc */
}

```

## V. PHÂN TÍCH TỪ VỰNG (Lexical Analysis)

Bây giờ chúng ta thêm vào phần trước trình biên dịch một bộ phân tích từ vựng để đọc và biến đổi dòng nhập thành một chuỗi các từ tố (token) mà bộ phân tích cú pháp có thể sử dụng được. Nhắc lại rằng một chuỗi các ký tự hợp thành một token gọi là từ vựng (lexeme) của token đó.

Trước hết ta trình bày một số chức năng cần thiết của bộ phân tích từ vựng.

### 1. Loại bỏ các khoảng trắng và các dòng chú thích

Quá trình dịch sẽ xem xét tất cả các ký tự trong dòng nhập nên những ký tự không có nghĩa (như khoảng trắng bao gồm ký tự trống (blanks), ký tự tabs, ký tự newlines)

hoặc các dòng chú thích (comment) phải bị bỏ qua. Khi bộ phân tích từ vựng đã bỏ qua các khoảng trắng này thì bộ phân tích cú pháp không bao giờ xem xét đến chúng nữa. Chọn lựa cách sửa đổi văn phạm để đưa cả khoảng trắng vào trong cú pháp thì hầu như rất khó cài đặt.

## 2. Xử lý các hằng

Bất cứ khi nào một ký tự số xuất hiện trong biểu thức thì nó được xem như là một hằng số. Bởi vì một hằng số nguyên là một dãy các chữ số nên nó có thể được cho bởi luật sinh văn phạm hoặc tạo ra một token cho hằng số đó. Bộ phân tích từ vựng có nhiệm vụ ghép các chữ số để được một số và sử dụng nó như một đơn vị trong suốt quá trình dịch.

Đặt **num** là một token biểu diễn cho một số nguyên. Khi một chuỗi các chữ số xuất hiện trong dòng nhập thì bộ phân tích từ vựng sẽ gửi **num** cho bộ phân tích cú pháp. Giá trị của số nguyên được chuyển cho bộ phân tích cú pháp như là một thuộc tính của token **num**. Về mặt logic, bộ phân tích từ vựng sẽ chuyển cả token và các thuộc tính cho bộ phân tích cú pháp. Nếu ta viết một token và thuộc tính thành một bộ nằm giữa  $< >$  thì dòng nhập  $31 + 28 + 59$  sẽ được chuyển thành một dãy các bộ :

$<\text{num}, 31>, <+, >, <\text{num}, 28>, <+, >, <\text{num}, 59>.$

Bộ  $<+, >$  cho thấy thuộc tính của  $+$  không có vai trò gì trong khi phân tích cú pháp nhưng nó cần thiết dùng đến trong quá trình dịch.

## 3. Nhận dạng các danh biểu và từ khóa

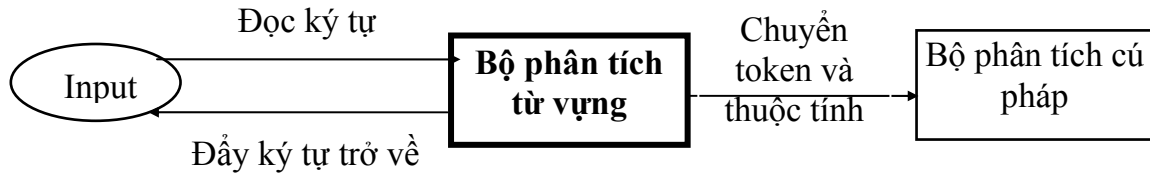
Ngôn ngữ dùng các danh biểu (identifier) như là tên biến, mảng, hàm và văn phạm xử lý các danh biểu này như là một token. Người ta dùng token id cho các danh biểu khác nhau do đó nếu ta có dòng nhập  $count = count + increment;$  thì bộ phân tích từ vựng sẽ chuyển cho bộ phân tích cú pháp chuỗi token: **id** = **id** + **id** (cần phân biệt token và trị từ vựng lexeme của nó: token id nhưng trị từ vựng (lexeme) có thể là *count* hoặc *increment*). Khi một lexeme thể hiện cho một danh biểu được tìm thấy trong dòng nhập cần phải có một cơ chế để xác định xem lexeme này đã được thấy trước đó chưa? Công việc này được thực hiện nhờ sự lưu trữ trợ giúp của *bảng ký hiệu* (symbol table) đã nêu ở chương trước. Trị từ vựng được lưu trong bảng ký hiệu và một con trỏ chỉ đến mục ghi trong bảng trở thành một thuộc tính của token **id**.

Nhiều ngôn ngữ cũng sử dụng các chuỗi ký tự cố định như **begin**, **end**, **if**, ... để xác định một số kết cấu. Các chuỗi ký tự này được gọi là từ khóa (keyword). Các từ khóa cũng thỏa mãn qui luật hình thành danh biểu, do vậy cần qui ước rằng một chuỗi ký tự được xác định là một danh biểu khi nó không phải là từ khóa.

Một vấn đề nữa cần quan tâm là vấn đề tách ra một token trong trường hợp một ký tự có thể xuất hiện trong trị từ vựng của nhiều token. Ví dụ một số các token là các toán tử quan hệ trong Pascal như :  $<, <=, <>.$

## 4. Giao diện của bộ phân tích từ vựng

Bộ phân tích từ vựng được đặt xen giữa dòng nhập và bộ phân tích cú pháp nên giao diện với hai bộ này như sau:



**Hình 2.12** - *Giao diện của bộ phân tích từ vựng*

Bộ phân tích từ vựng đọc từng ký tự từ dòng nhập, nhóm chúng lại thành các trị từ vựng và chuyển các token xác định bởi trị từ vựng này cùng với các thuộc tính của nó đến những giai đoạn sau của trình biên dịch. Trong một vài tình huống, bộ phân tích từ vựng phải đọc vượt trước một số ký tự mới xác định được một token để chuyển cho bộ phân tích cú pháp. Ví dụ, trong Pascal khi gặp ký tự >, phải đọc thêm một ký tự sau đó nữa; nếu ký tự sau là = thì token được xác định là “lớn hơn hoặc bằng”, ngược lại thì token là “lớn hơn” và do đó một ký tự đã bị đọc quá. Trong trường hợp đó thì ký tự đọc quá này phải được đẩy trả về (push back) cho dòng nhập vì nó có thể là ký tự bắt đầu cho một trị từ vựng mới.

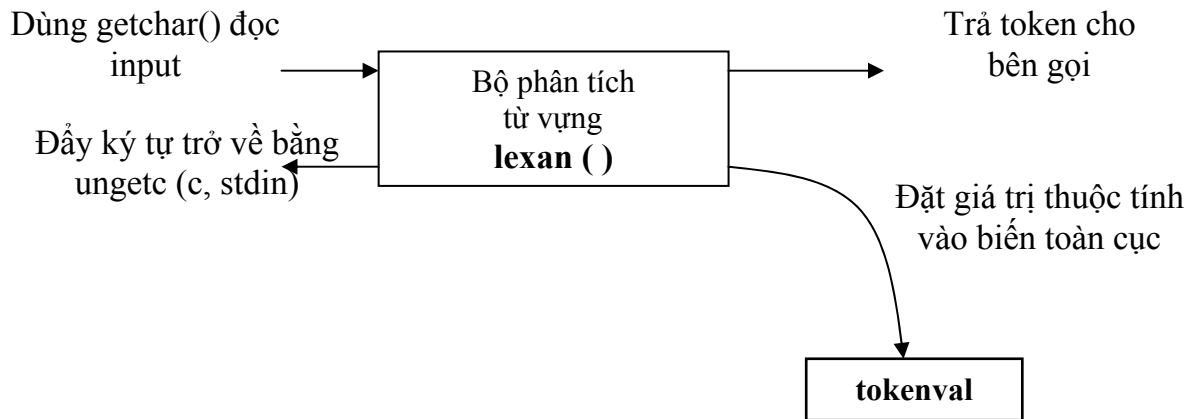
Bộ phân tích từ vựng và bộ phân tích cú pháp tạo thành một cặp "nhà sản xuất - người tiêu dùng" (producer - consumer). Bộ phân tích từ vựng sản sinh ra các token và bộ phân tích cú pháp tiêu thụ chúng. Các token được sản xuất bởi bộ phân tích từ vựng sẽ được lưu trong một bộ đệm (buffer) cho đến khi chúng được tiêu thụ bởi bộ phân tích cú pháp. Bộ phân tích từ vựng không thể hoạt động tiếp nếu buffer bị đầy và bộ phân tích cú pháp không thể hoạt động nữa nếu buffer rỗng. Thông thường, buffer chỉ lưu giữ một token. Để cài đặt tương tác dễ dàng, người ta tạo ra một thủ tục phân tích từ vựng được gọi từ trong thủ tục phân tích cú pháp, mỗi lần gọi trả về một token.

Việc đọc và quay lui ký tự cũng được cài đặt bằng cách dùng một bộ đệm nhập. Một khối các ký tự được đọc vào trong buffer nhập tại một thời điểm nào đó, một con trỏ sẽ giữ vị trí đã được phân tích. Quay lui ký tự được thực hiện bằng cách lùi con trỏ trở về. Các ký tự trong dòng nhập cũng có thể cần được lưu lại cho công việc ghi nhận lỗi bởi vì cần phải chỉ ra vị trí lỗi trong đoạn chương trình.

Để tránh việc phải quay lui, một số trình biên dịch sử dụng cơ chế đọc trước một ký tự rồi mới gọi đến bộ phân tích từ vựng. Bộ phân tích từ vựng sẽ đọc tiếp các ký tự cho đến khi đọc tới ký tự mở đầu cho một token khác. Trị từ vựng của token trước đó sẽ bao gồm các ký tự từ ký tự đọc trước đến ký tự vừa ngay ký tự vừa đọc được. Ký tự vừa đọc được sẽ là ký tự mở đầu cho trị từ vựng của token sau. Với cơ chế này thì mỗi ký tự chỉ được đọc duy nhất một lần.

## 5. Một bộ phân tích từ vựng

Bây giờ chúng ta xây dựng một bộ phân tích từ vựng cho chương trình dịch các biểu thức số học. Hình sau đây gợi ý một cách cài đặt giao diện của bộ phân tích từ vựng được viết bằng C dưới dạng hàm lexan. Lexan đọc và đẩy các ký tự trong input trở về bằng cách gọi thủ tục getch và ungetc.



**Hình 2.13** - Cài đặt giao diện của bộ phân tích từ vựng

Nếu ngôn ngữ cài đặt không cho phép trả về các cấu trúc dữ liệu từ các hàm thì token và các thuộc tính của nó phải được truyền riêng rẽ. Hàm `lexan` trả về một số nguyên mã hóa cho một token. Token cho một ký tự có thể là một số nguyên quy ước được dùng để mã hóa cho ký tự đó. Một token như `num` có thể được mã hóa bằng một số nguyên lớn hơn mọi số nguyên được dùng để mã hóa cho các ký tự, chẳng hạn là 256. Để dễ dàng thay đổi cách mã hóa, chúng ta dùng một hằng tượng trưng `NUM` thay cho số nguyên mã hóa của `num`. Hàm `lexan` trả về `NUM` khi một dãy chữ số được tìm thấy trong input. Biến toàn cục `tokenval` được đặt là giá trị của chuỗi số này.

Cài đặt của hàm `lexan` như sau :

```

#include<stdio.h>
#include<ctype.h>
int  lineno = 1;
int  tokenval = NONE;
int  lexan ( )
{
    int  t;
    while(1) {
        t = getchar();
        if ( t == ' ' || t == '\t' ) ;      /* loại bỏ blank và tab */
        else if ( t == '\n' )
            lineno = lineno + 1;
        else if ( isdigit(t) ) {
            tokenval = t - '0';
            t = getchar();
            while ( isdigit(t) ) {
                tokenval = tokenval * 10 + t - '0';
                t = getchar();
            }
        }
    }
}
  
```

```

    }
    ungetc (t, stdin);
    return NUM;
}
else {
    tokenval = NONE;
    return t;
}
} /* while */
} /* lexan */

```

## VI. SỰ HÌNH THÀNH BẢNG KÝ HIỆU

Một cấu trúc dữ liệu gọi là *bảng ký hiệu* (symbol table) thường được dùng để lưu giữ thông tin về các cấu trúc của ngôn ngữ nguồn. Các thông tin này được tập hợp từ các giai đoạn phân tích của trình biên dịch và được sử dụng bởi giai đoạn tổng hợp để sinh mã đích. Ví dụ trong quá trình phân tích từ vựng, các chuỗi ký tự tạo ra một token (trị từ vựng của token) sẽ được lưu vào một mục ghi trong bảng danh biểu. Các giai đoạn sau đó có thể bổ sung thêm các thông tin về kiểu của danh biểu, cách sử dụng nó và vị trí lưu trữ. Giai đoạn sinh mã sẽ dùng thông tin này để tạo ra mã phù hợp, cho phép lưu trữ và truy xuất biến đó.

### 1. Giao diện của bảng ký hiệu

Các thủ tục trên bảng ký hiệu chủ yếu liên quan đến việc lưu trữ và truy xuất các trị từ vựng. Khi một trị từ vựng được lưu trữ thì token kết hợp với nó cũng được lưu. Hai thao tác sau được thực hiện trên bảng ký hiệu.

**Insert (s, t):** Trả về chỉ mục của một ô mới cho chuỗi s, token t.

**Lookup (s):** Trả về chỉ mục của ô cho chuỗi s hoặc 0 nếu chuỗi s không tồn tại.

Bộ phân tích từ vựng sử dụng thao tác tìm kiếm **lookup** để xác định xem một ô cho một trị từ vựng của một token nào đó đã tồn tại trong bảng ký hiệu hay chưa? Nếu chưa thì dùng thao tác xen vào **insert** để tạo ra một ô mới cho nó.

### 2. Xử lý từ khóa dành riêng

Ta cũng có thể sử dụng bảng ký hiệu nói trên để xử lý các từ khóa dành riêng (reserved keyword). Ví dụ với hai token **div** và **mod** với hai trị từ vựng tương ứng là div và mod. Chúng ta có thể khởi tạo bảng ký hiệu bởi lời gọi:

```
insert ("div", div);
```

```
insert ("mod", mod);
```

Sau đó lời gọi lookup ("div") sẽ trả về token **div**, do đó "div" không thể được dùng làm danh biểu.

Với phương pháp vừa trình bày thì tập các từ khóa được lưu trữ trong bảng ký hiệu trước khi việc phân tích từ vựng diễn ra. Ta cũng có thể lưu trữ các từ khóa bên ngoài

bảng ký hiệu như là một danh sách có thứ tự của các từ khóa. Trong quá trình phân tích từ vựng, khi một trị từ vựng được xác định thì ta phải tìm (nhị phân) trong danh sách các từ khóa xem có trị từ vựng này không. Nếu có, thì trị từ vựng đó là một từ khóa, ngược lại, đó là một danh biểu và sẽ được đưa vào bảng ký hiệu.

### 3. Cài đặt bảng ký hiệu

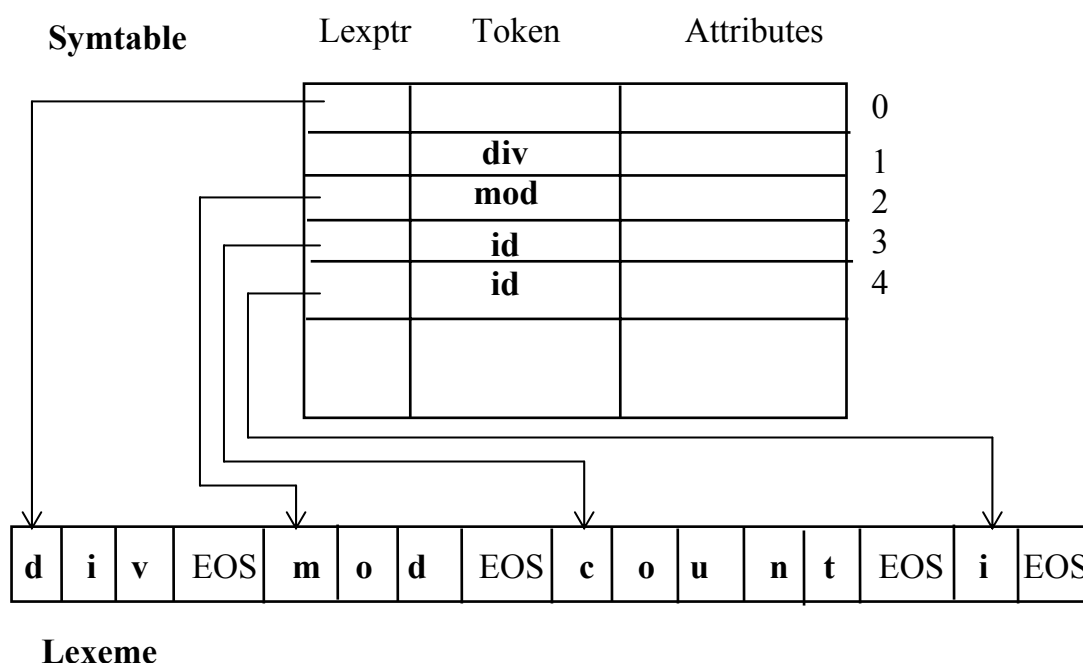
Cấu trúc dữ liệu cụ thể dùng cài đặt cho một bảng ký hiệu được trình bày trong hình dưới đây. Chúng ta không muốn dùng một lượng không gian nhớ nhất định để lưu các trị từ vựng tạo ra một danh biểu bởi vì một lượng không gian cố định có thể không đủ lớn để lưu các danh biểu rất dài và cũng rất lãng phí khi gặp một danh biểu ngắn.

Thông thường, một bảng ký hiệu gồm hai mảng :

1. Mảng **lexemes** (trị từ vựng) dùng để lưu trữ các chuỗi ký tự tạo ra một danh biểu, các chuỗi này ngăn cách nhau bởi các ký tự EOS (end - of - string).

2. Mảng **symtable** với mỗi phần tử là một mẫu tin (record) bao gồm hai trường, trường con trỏ **lexptr** trỏ tới đầu trị từ vựng và trường **token**. Cũng có thể dùng thêm các trường khác để lưu trữ giá trị các thuộc tính.

Mục ghi thứ zero trong mảng symtable phải được để trống bởi vì giá trị trả về của hàm lookup trong trường hợp không tìm thấy ô tương ứng cho chuỗi ký hiệu.



**Hình 2.14 - Bảng ký hiệu và mảng để lưu các chuỗi**

Trong hình trên, ô thứ nhất và thứ hai trong bảng ký hiệu dành cho các từ khóa **div** và **mod**. Ô thứ ba và thứ tư dành cho các danh biểu **count** và **i**.

Đoạn mã (ngôn ngữ giả) cho bộ phân tích từ vựng được dùng để xử lý các danh biểu như sau. Nó xử lý khoảng trắng và hằng số nguyên cũng giống như thủ tục đã nói ở phần trước. Khi bộ phân tích từ vựng đọc vào một chữ cái, nó bắt đầu lưu các chữ cái và chữ số vào trong vùng đệm lexbuf. Chuỗi được tập hợp trong lexbuf sau đó được tìm trong mảng symtable của bảng ký hiệu bằng cách dùng hàm **lookup**. Bởi vì bảng ký hiệu đã được khởi tạo với 2 ô cho div và mod (hình 2.14) nên nó sẽ tìm thấy

trị từ vựng này nếu lexbuf có chứa div hay mod, ngược lại nếu không có ô cho chuỗi đang chứa trong lexbuf thì hàm **lookup** sẽ trả về 0 và do đó hàm **insert** được gọi để tạo ra một ô mới trong symtable và p là chỉ số của ô trong bảng ký hiệu của chuỗi trong lexbuf. Chỉ số này được truyền tới bộ phân tích cú pháp bằng cách đặt tokenval := p và token nằm trong trường token được trả về.

Kết quả mặc nhiên là trả về số nguyên mã hóa cho ký tự dùng làm token.

**Function** lexan: integer;

var lexbuf: array[0..100] of char;

c: char

**begin**

**loop begin**

đọc một ký tự vào c;

**if** c là một ký tự trống blank hoặc ký tự tab **then**

không thực hiện điều gì ;

**else if** c là ký tự newline **then**

lineno = lineno + 1

**else if** c là một ký tự số **then**

**begin**

đặt tokenval là giá trị của ký số này và các ký số theo sau;

**return** NUM;

**end**

**else if** c là một chữ cái **then**

**begin**

đặt c và các ký tự, ký số theo sau vào lexbuf;

p := **lookup** (lexbuf);

**if** p = 0 **then** p := insert (lexbuf, id);

tokenval := p;

**return** trường token của ô có chỉ mục p;

**end**

**else**

**begin** /\* token là một ký tự đơn \*/

đặt tokenval là NONE; /\* không có thuộc tính \*/

**return** số nguyên mã hóa của ký tự c;

**end;**

**end;**



end;

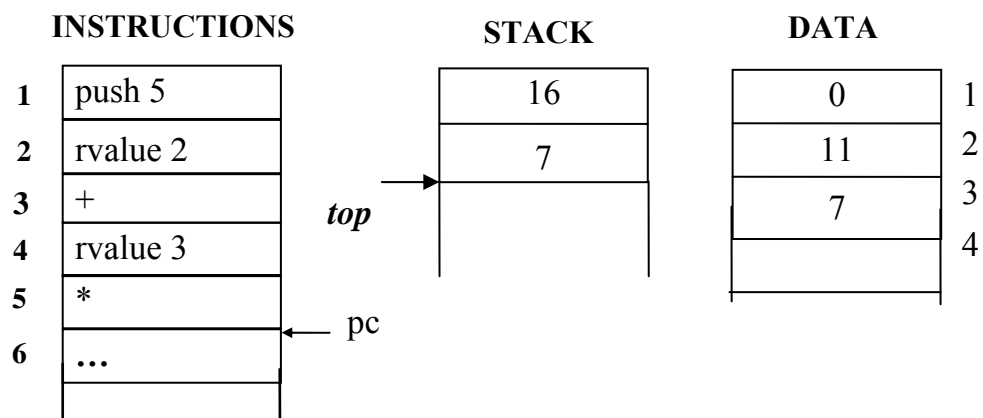
## VII. MÁY ẢO KIỂU STACK

Ta đã biết rằng kết quả của giai đoạn phân tích là một biểu diễn trung gian của chương trình nguồn mà giai đoạn tổng hợp sử dụng nó để phát sinh mã đích. Một dạng phổ biến của biểu diễn trung gian là mã của một máy ảo kiểu Stack (abstract stack machine - ASM).

Trong phần này, chúng ta sẽ trình bày khái quát về một máy ảo kiểu Stack và chỉ ra cách sinh mã chương trình cho nó. Máy ảo này bao gồm 3 thành phần:

1. Vùng nhớ chỉ thị (instructions): là nơi chứa các chỉ thị. Các chỉ thị này rất hạn chế và được chia thành 3 nhóm chính: nhóm chỉ thị số học trên số nguyên, nhóm chỉ thị thao tác trên Stack và nhóm chỉ thị điều khiển trình tự.
2. Vùng Stack: là nơi thực hiện các chỉ thị trên các phép toán số học.
3. Vùng nhớ dữ liệu (data): là nơi lưu trữ riêng các dữ liệu.

Hình sau đây minh họa cho nguyên tắc thực hiện của dạng máy này, con trỏ pc (program counter) chỉ ra chỉ thị đang chờ để thực hiện tiếp theo. Các giá trị dùng trong quá trình tính toán được nạp vào đỉnh Stack. Sau khi tính toán xong, kết quả được lưu tại đỉnh Stack.



**Hình 2.15** - Minh họa hình ảnh một máy ảo kiểu Stack

**Ví dụ 2.15:** Biểu thức  $(5 + b) * c$  với  $b = 11$ ,  $c = 7$  sẽ được thực hiện trên Stack dưới dạng biểu thức hậu tố  $5\ b\ +\ c\ *$ .

### 1. Các chỉ thị số học

Máy ảo phải cài đặt mỗi toán tử bằng một ngôn ngữ trung gian. Khi gặp các chỉ thị số học đơn giản, máy sẽ thực hiện phép toán tương ứng với hai giá trị trên đỉnh Stack, kết quả cũng được lưu vào đỉnh STACK. Một phép toán phức tạp hơn có thể cần phải được cài đặt như một loạt chỉ thị của máy.

Mã chương trình máy ảo cho một biểu thức số học sẽ mô phỏng hành động ước lượng dạng hậu tố cho biểu thức đó bằng cách sử dụng Stack. Việc ước lượng được tiến hành bằng cách xử lý chuỗi hậu tố từ trái sang phải, đẩy mỗi toán hạng vào Stack khi gặp nó. Với một toán tử  $k$  - ngôi, đối số cận trái của nó nằm ở  $(k - 1)$  vị trí bên dưới đỉnh Stack và đối số cận phải nằm tại đỉnh. Hành động ước lượng áp dụng toán tử cho  $k$  giá trị trên đỉnh của Stack, lấy toán hạng ra và đặt kết quả trở lại vào Stack.

Trong ngôn ngữ trung gian, mọi giá trị đều là số nguyên; số 0 tương ứng với false và các số khác 0 tương ứng với true. Toán tử logic **and** và **or** cần phải có cả 2 đối số.

## 2. Chỉ thị L- value và R-value

Ta cần phân biệt ý nghĩa của các danh biểu ở vế trái và vế phải của một phép gán. Trong mỗi phép gán sau :

$i := 5;$

$i := i + 1;$

vế phải xác định một giá trị nguyên, còn vế trái xác định nơi giá trị được lưu. Tương tự, nếu p và q là những con trỏ đến các ký tự dạng :

$p \uparrow := q \uparrow;$

thì vế phải  $q \uparrow$  xác định một ký tự, còn  $p \uparrow$  xác định vị trí ký tự được lưu. Các thuật ngữ L-value (giá trị trái) và R-value (giá trị phải) muốn nói đến các giá trị thích hợp tương ứng ở vế trái và vế phải của một phép gán. Nghĩa là, R-value có thể được xem là 'giá trị' còn L-value chính là các địa chỉ.

**L-value l** : Đẩy nội dung ở vị trí dữ liệu l vào Stack

**R-value l** : Đẩy địa chỉ của vị trí dữ liệu l vào Stack

## 3. Các chỉ thị thao tác trên STACK

Bên cạnh những chỉ thị cho thao tác đẩy một hằng số nguyên vào Stack và lấy một giá trị ra khỏi đỉnh Stack, còn có một số chỉ thị truy xuất vùng nhớ dữ liệu như sau:

**push v** : Đẩy giá trị v vào đỉnh Stack ( $top := top + 1$ )

**pop** : Lấy giá trị ra khỏi đỉnh Stack ( $top := top + 1$ )

**:=** : R-value trên đỉnh Stack được lưu vào L-value ngay bên dưới nó và lấy cả hai ra khỏi Stack ( $top := top - 2$ )

**copy** : Sao chép giá trị tại đỉnh Stack ( $top := top + 1$ )

## 4. Dịch các biểu thức

Đoạn mã chương trình dùng để ước lượng một biểu thức trên một máy ảo kiểu Stack có liên quan mật thiết với ký pháp hậu tố cho biểu thức đó.

**Ví dụ 2.16:** Dịch phép gán sau thành mã máy ảo kiểu Stack:

$day := (1461 * y) \text{ div } 4 + (153 * m + 2) \text{ div } 5 + d$

Ký pháp hậu tố của biểu thức như sau :

$day \ 1461 \ y * 4 \text{ div } 153 \ m * 2 + 5 \text{ div } + d + :=$

Đoạn mã máy có dạng :

L-value	day	push	2
push	1461	+	
R-value	y	push	5
*		div	
push	4	+	

div		R-value	d
push	153	+	
R- value	m	:=	
*			

## 5. Các chỉ thị điều khiển trình tự

Máy ảo kiểu Stack thực hiện các chỉ thị theo đúng thứ tự liệt kê trừ khi được yêu cầu thực hiện khác đi bằng các câu lệnh nhảy có điều kiện hoặc không điều kiện. Có một số các tùy chọn dùng để mô tả các đích nhảy :

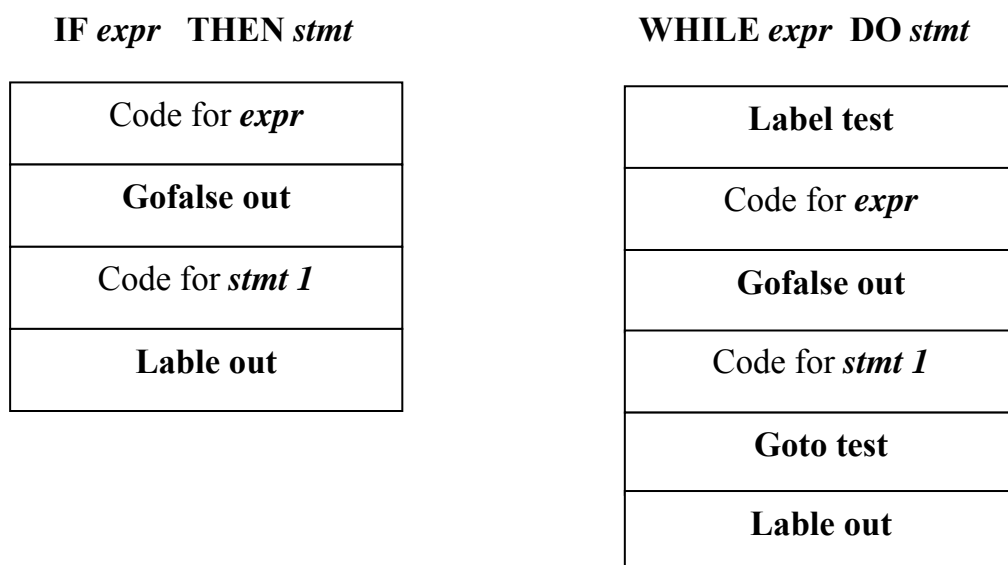
1. Toán hạng làm chỉ thị cho biết vị trí đích.
2. Toán hạng làm chỉ thị mô tả khoảng cách tương đối cần nhảy theo chiều tới hoặc lui.
3. Đích nhảy đến được mô tả bằng các ký hiệu tượng trưng gọi là các nhãn.

Một số chỉ thị điều khiển trình tự cho máy là :

- lable l** : Gán đích của các lệnh nhảy đến là l, không có tác dụng khác.
- goto l** : Chỉ thị tiếp theo được lấy từ câu lệnh có lable l .
- gofalse l** : Lấy giá trị trên đỉnh Stack ra, nếu giá trị là 0 thì nhảy đến l, ngược lại, thực hiện lệnh kế tiếp.
- gotrue l** : Lấy giá trị trên đỉnh Stack ra, nếu giá trị khác 0 thì nhảy đến l, ngược lại, thực hiện lệnh kế tiếp.
- halt** : Ngưng thực hiện chương trình.

## 6. Dịch các câu lệnh

Sơ đồ phác thảo đoạn mã máy ảo cho một số lệnh cấu trúc được chỉ ra trong hình sau:



**Hình 2.16** - Sơ đồ đoạn mã cho một số lệnh cấu trúc

Xét sơ đồ đoạn mã cho câu lệnh If . Giả sử rằng newlable là một thủ tục trả về một

nhãn mới cho mỗi lần gọi. Trong hành vi ngữ nghĩa sau đây, nhãn được trả về bởi một lời gọi đến newlabel được ghi lại bằng cách dùng một biến cục bộ out :

```
stmt → if expr then stmt1      { out := newlabel;
                                stmt.t := expr.t ||
                                ' gofalse ' out ||
                                stmt1.t ||
                                ' lable ' out      }
```

Thay vì in ra các câu lệnh, ta có thể sử dụng thủ tục emit để che dấu các chi tiết in. Chẳng hạn như emit phải xem xét xem mỗi chỉ thị máy ảo có cần nằm trên một hàng riêng biệt hay không. Sử dụng thủ tục emit, ta có thể viết lại như sau :

```
stmt → if
      expr      { out := newlabel; emit ( ' gofalse ', out); }
      then
      stmt1      { emit ( ' lable ', out); }
```

Khi một hành vi ngữ nghĩa xuất hiện bên trong một luật sinh, ta xét các phần tử ở vế phải của luật sinh theo thứ tự từ trái sang phải. Đoạn mã (ngôn ngữ giả) cho phép dịch phép gán và câu lệnh điều kiện If tương ứng như sau :

```
procedure stmt;
  var test, out: integer;      /* dùng cho các nhãn */
begin
  if lookahead = id then
    begin
      emit ('lvalue', tokenval); match (id); match (':='); expr;
    end
  else if lookahead = 'if' then
    begin
      match ('if'); expr; out := newlabel;
      emit ('gofalse', out); match ('then'); stmt;
      emit ('lable', out);
    end
    /* đoạn mã cho các lệnh còn lại */
  else error;
end;
```

## VIII. KẾT NỐI CÁC KỸ THUẬT

Trong các phần trên, chúng ta đã trình bày một số kỹ thuật biên dịch trực tiếp cú pháp để xây dựng kỳ đầu của trình biên dịch. Phần này sẽ thực hiện việc kết nối chúng lại bằng cách giới thiệu một chương trình C có chức năng dịch trung tố - hậu tố cho một ngôn ngữ gồm dãy các biểu thức kết thúc bằng các dấu chấm phẩy. Các biểu thức gồm có các số, danh biểu, các toán tử +, -, \*, /, div và mod. Output cho chương trình là dạng biểu diễn hậu tố cho mỗi biểu thức.

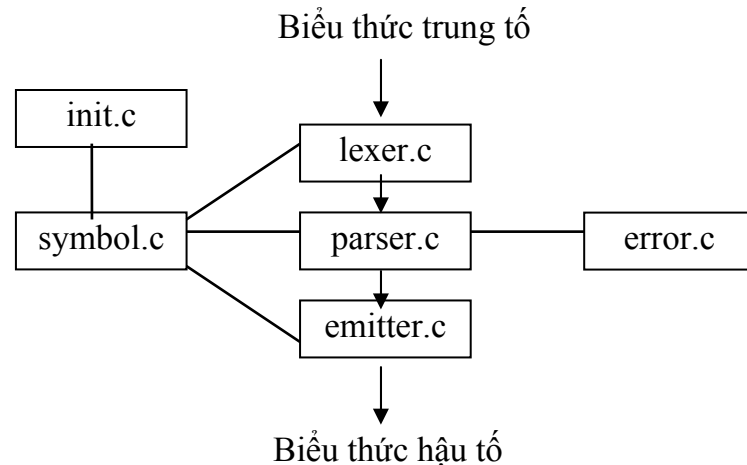
### 1. Mô tả chương trình dịch

Chương trình dịch được thiết kế bằng cách dùng lược đồ dịch trực tiếp cú pháp có dạng như sau :

```
start → list eof
list  → expr ; list
      | ε
expr → expr + term    { print ( '+' ) }
      | expr - term    { print ( '-' ) }
      | term
term → term * factor   { print ( '*' ) }
      | term / factor   { print ( '/' ) }
      | term div factor { print ( 'DIV' ) }
      | term mod factor { print ( 'MOD' ) }
      | factor
factor → ( expr )
       | id           { print ( id.lexeme ) }
       | num          { print ( num.value ) }
```

Trong đó, token **id** biểu diễn một dãy không rỗng gồm các chữ cái và ký số bắt đầu bằng một chữ cái, **num** là dãy ký số, **eof** là ký tự cuối tập tin (end - of - file). Các token được phân cách bởi một dãy ký tự blank, tab và newline - gọi chung là các khoảng trắng (white space). Thuộc tính *lexeme* của token **id** là chuỗi ký tự tạo ra token đó, thuộc tính *value* của token **num** chứa số nguyên được biểu diễn bởi **num**.

Đoạn mã cho chương trình dịch bao gồm 7 thủ tục, mỗi thủ tục được lưu trong một tập tin riêng. Điểm bắt đầu thực thi chương trình nằm trong thủ tục chính **main.c** gồm có một lời gọi đến **init( )** để khởi gán, theo sau là một lời gọi đến **parse( )** để dịch. Các thủ tục còn lại được mô tả tổng quan như hình sau:



**Hình 2.17** - Sơ đồ các thủ tục cho chương trình dịch biểu thức

Trước khi trình bày đoạn mã lệnh cho chương trình dịch, chúng ta mô tả sơ lược từng thủ tục và cách xây dựng chúng.

### Thủ tục phân tích từ vựng lexer.c

Bộ phân tích từ vựng là một thủ tục có tên **lexan()** được gọi từ bộ phân tích cú pháp khi cần tìm các token. Thủ tục này đọc từng ký tự trong dòng nhập, trả về token vừa xác định được cho bộ phân tích cú pháp. Giá trị của các thuộc tính đi kèm với token được gán cho biến toàn cục tokenval. Bộ phân tích cú pháp có thể nhận được các token sau : + - \* / DIV MOD ( ) ID NUM DONE

Trị từ vựng	Token	Giá trị thuộc tính
Khoảng trắng		
Chuỗi các chữ số	NUM	Giá trị số
Div	DIV	
Mod	MOD	
Chuỗi mở đầu là chữ cái, theo sau là chữ cái hoặc chữ số	ID	Chỉ số trong symtable
Ký tự cuối tập tin - eof	DONE	
Các ký tự khác	Ký tự tương ứng	NONE

Trong đó ID biểu diễn cho một danh biểu, NUM biểu diễn cho một số và DONE là ký tự cuối tập tin eof. Các khoảng trắng đã được loại bỏ. Bảng sau trình bày các token và giá trị thuộc tính được sinh ra bởi bộ phân tích từ vựng cho mỗi token trong chương trình nguồn.

### Thủ tục phân tích cú pháp parser.c

Bộ phân tích cú pháp được xây dựng theo phương pháp phân tích đệ quy xuống. Trước tiên, ta loại bỏ đệ quy trái ra khỏi lược đồ dịch bằng cách thêm vào 2 biến mới R1 cho expr và R2 cho factor, thu được lược đồ dịch mới như sau:

start → list eof

```

list → expr ; list
      | ε
expr → term R1
R1  → + term { print ( ' + ' ) } R1
      | - term { print ( ' - ' ) } R1
      | ε
term → factor R2
R2  → * factor { print ( ' * ' ) } R2
      | / factor { print ( ' / ' ) } R2
      | DIV factor { print ( 'DIV' ) } R2
      | MOD factor { print ( 'MOD' ) } R2
      | ε
factor → ( expr )
        | id { print ( id.lexeme ) }
        | num { print ( num.value ) }

```

Sau đó, chúng ta xây dựng các hàm cho các ký hiệu chưa kết thúc *expr*, *term* và *factor*. Hàm **parse( )** cài đặt ký hiệu bắt đầu start của văn phạm, nó gọi **lexan** mỗi khi cần một token mới. Bộ phân tích cú pháp ở giai đoạn này sử dụng hàm **emit** để sinh ra kết quả và hàm **error** để ghi nhận một lỗi cú pháp.

### Thủ tục kết xuất emitter.c

Thủ tục này chỉ có một hàm **emit (t, tval)** sinh ra kết quả cho token *t* với giá trị thuộc tính *tval*.

### Thủ tục quản lý bảng ký hiệu symbol.c và khởi tạo init.c

Thủ tục **symbol.c** cài đặt cấu trúc dữ liệu cho bảng danh biểu. Các ô trong mảng *symtable* là các cặp gồm một con trỏ chỉ đến mảng *lexemes* và một số nguyên biểu thị cho token được lưu tại vị trí đó.

Thủ tục **init.c** được dùng để khởi gán các từ khóa vào bảng danh biểu. Biểu diễn trị từ vựng và token cho tất cả các từ khóa được lưu trong mảng *keywords* cùng kiểu với mảng *symtable*. Hàm **init( )** duyệt lần lượt qua mảng *keyword*, sử dụng hàm **insert** để đặt các từ khóa vào bảng danh biểu.

### Thủ tục lỗi error.c

Thủ tục này quản lý các ghi nhận lỗi và hết sức cần thiết. Khi gặp một lỗi cú pháp, trình biên dịch in ra một thông báo cho biết rằng một lỗi đã xảy ra trên dòng nhập hiện hành và dừng lại. Một kỹ thuật khắc phục lỗi tốt hơn có thể sẽ nhảy qua dấu chấm phẩy kế tiếp và tiếp tục phân tích câu lệnh sau đó.

## 2. Cài đặt chương trình nguồn

Chương trình nguồn C cài đặt chương trình dịch trên.

```

/ ****      global.h      *****/

#include <stdio.h>      /* tải các thủ tục xuất nhập */
#include <ctype.h>      /* tải các thủ tục kiểm tra ký tự */

#define BSIZE 128      /* buffer size kích thước vùng đệm */
#define NONE -1
#define EOS '\0'

#define NUM 256
#define DIV 257
#define MOD 258
#define ID 259
#define DONE 260

int tokenval;          /* giá trị của thuộc tính token */
int lineno;

struct entry {          /* khuôn dạng cho ô trong bảng ký hiệu */
    char * lexptr;
    int token;
}
struct entry symtable[ ] /* bảng ký hiệu */

/ ****      lexer.c      *****/

#include "global.h"

char lexbuf[BSIZE]
int lineno = 1;
int tokenval = NONE;

int lexan ( )          /* bộ phân tích từ vựng */
{
    int t;
    while(1) {
        t = getchar ( );
        if ( t == ' ' || t == '\t' ) ;      /* xóa các khoảng trắng */
        else if ( t == '\n' )
            lineno = lineno + 1;
        else if ( isdigit (t) ) {          /* t là một ký số */
            ungetc (t, stdin);
            scanf ("%d", & tokenval);
            return NUM;
        }
        else if ( isalpha (t) ) {          /* t là một chữ cái */

```



```

        int p, b = 0;
        while ( isalnum (t) ) { /* t thuộc loại chữ - số */
            lexbuf[b] = t;
            t = getchar ( );
            b = b + 1;
            if (b >= BSIZE)
                error("compiler error");
        }
        lexbuf[b] = EOS;
        if (t != EOF)
            ungetc (t, stdin);
        p = lookup (lexbuf);
        if (p == 0)
            p = insert (lexbuf, ID)
        tokenval = p;
        return symtable[p].token;
    }
    else if (t == EOF) {
        return DONE;
    }
    else {
        tokenval = NONE;
        return t;
    }
}

/ ****      parser.c      *****/

#include "global.h"

int lookahead;

parse ( ) /* phân tích cú pháp và dịch danh sách biểu thức */
{
    lookahead = lexan ( );
    while (lookahead != DONE) {
        expr ( ); match ( ' ; ');
    }
}

expr ( )
{
    int t;
    term ( );
    while(1)
        switch (lookahead) {
            case ' + ': case ' - ':
                t = lookahead;

```

```

        match (lookahead); term ( ); emit (t, NONE);
        continue;
    default :
        return;
    }
}

term ( )
{
    int    t;
    factor ( );
    while(1)
        switch (lookahead)    {
            case '*' : case '/' : case 'DIV' : case 'MOD' :
                t = lookahead;
                match (lookahead); factor ( ); emit (t, NONE);
                continue;
            default :
                return;
        }
}

factor ( )
{
    switch (lookahead)    {
        case '(' :
            match ( '(' ); expr ( ); match ( ')' );    break;
        case NUM :
            emit (NUM, tokenval) ; match ( ' NUM ' );    break;
        case ID :
            emit (ID, tokenval) ; match ( ' ID ' );    break;
        default :
            error ( "syntax error");
    }
}

match ( t )
    int t;
{
    if (lookahead == t)
        lookahead = lexan ( );
    else error ("syntax error");
}

/ ****          emitter.c          *****/

# include "global.h"

```

```

emit (t, tval)          /* tạo ra kết quả */
    int    t, tval;
{
    switch ( t )    {
        case '+' :      case '-' :      case '*' :      case '/' :
            printf (" %c \n", t); break;
        case DIV :
            printf (" DIV \n", t);      break;
        case MOD :
            printf (" MOD \n", t);      break;
        case NUM :
            printf (" %d \n", tval );    break;
        case ID :
            printf (" %s \n", symtable [tval]. lexptr);    break;
        default :
            printf (" token %d , tokenval %d \n ", t, tval );
    }
}

/ ****          symbol.c          *****/

#include "global.h"

#define STRMAX  999          /* kích thước mảng lexemes */
#define SYMMAX  100         /* kích thước mảng symtable */

char lexemes [STRMAX];
int lastchar = -1          /* vị trí được dùng cuối cùng trong lexemes */
struct entry symtable [SYMMAX];
int lastentry = 0          /* vị trí được dùng cuối cùng trong symtable */

int lookup (s)              /* trả về vị trí của ô cho s */
    char s [ ];
{
    int p;
    for (p = lastentry; p > 0; p = p - 1)
        if (strcmp (symtable[p].lexptr, s ) == 0)
            return p;
    return 0;
}

int insert (s, tok)          /* trả về vị trí của ô cho s */
    char s [ ];
    int tok;
{
    int len;
    len = strlen (s)          /* strlen tính chiều dài của s */

```

```

        if ( lastentry + 1 >= SYMMAX)
            error ("symbol table full");
        if ( lastchar + len + 1 >= SYMMAX)
            error ("lexemes array full");
        lastentry = lastentry + 1;
        symable [lastentry].token = tok;
        symable [lastentry].lexptr = &lexemes [lastchar + 1];
        lastchar = lastchar + len + 1;
        strcpy (symable [lastentry].lexptr, s);
        return lastentry;
    }

/ ****          init.c          *****/

#include "global.h"

struct entry keyword [ ] = {
    "div", DIV
    "mod", MOD
    0,      0
}

init ( )          /* đưa các từ khóa vào symtable */
{
    struct entry * p ;
    for (p = keywords; p → token; p ++ )
        if (strcmp (symtable[p].lexptr, s ) == 0)
            insert (p → lexptr, p → token) ;
}

/ ****          error.c          *****/

#include "global.h"

error (m)          /* sinh ra tất cả các thông báo lỗi */
    char * m;
{
    fprintf (stderr, " line %d : %s \n", lineno, m)
    exit ( 1 )      /* kết thúc không thành công */
}

/ ****          main.c          *****/

#include "global.h"

main ( )
{

```

```
    init ( );  
    parse ( );  
    exit (0);      /* kết thúc thành công */  
}  
  
/ *****/
```

## BÀI TẬP CHƯƠNG II

2.1. Cho văn phạm phi ngữ cảnh sau:

$$S \rightarrow S S + \mid S S * \mid a$$

- a) Viết các luật sinh dẫn ra câu nhập:  $a a + a *$
- b) Xây dựng một cây phân tích cú pháp cho câu nhập trên?
- c) Văn phạm này sinh ra ngôn ngữ gì? Giải thích câu trả lời.

2.2. Ngôn ngữ gì sinh ra từ các văn phạm sau? Văn phạm nào là văn phạm mơ hồ?

- a)  $S \rightarrow 0 S 1 \mid 0 1$
- b)  $S \rightarrow + S S \mid - S S \mid a$
- c)  $S \rightarrow S ( S ) S \mid \in$
- d)  $S \rightarrow a S b S \mid b S a S \mid \in$
- e)  $S \rightarrow a \mid S + S \mid S S \mid S * \mid ( S )$

2.3. Xây dựng văn phạm phi ngữ cảnh đơn nghĩa cho các ngôn ngữ sau đây:

- a) Các biểu thức số học dưới dạng hậu tố.
- b) Danh sách danh biểu có tính kết hợp trái được phân cách bởi dấu phẩy.
- c) Danh sách danh biểu có tính kết hợp phải được phân cách bởi dấu phẩy.
- d) Các biểu thức số học của số nguyên và danh biểu với 4 phép toán hai ngôi :  $+, -, *, /$ .

2.4. Viết chỉ thị máy ảo kiểu Stack cho quá trình dịch các biểu thức sau sang dạng hậu tố:

- a)  $t := (a \bmod b) * 1998 - (2000 * c + 100) \operatorname{div} 4 + 1999$
- b)  $t := a_1 \bmod c_2 + (b_3 - 156 * d_4) \operatorname{div} 7 / 3$
- c)  $y := x + 100 z^3 t$

2.5. Xây dựng lược đồ dịch trực tiếp cú pháp để dịch một biểu thức số học từ dạng trung tố sang dạng hậu tố ( cho các phép toán 2 ngôi ).

- a) Xây dựng chương trình đổi mã hậu tố sang mã máy ảo kiểu Stack .
- b) Viết chương trình thực thi mã máy ảo .

2.6. Yêu cầu như bài 5 cho biểu thức số học ở dạng hậu tố sang dạng trung tố.

2.7. Xây dựng một lược đồ dịch trực tiếp cú pháp để xác định rằng các dấu ngoặc trong một chuỗi nhập là cân bằng.

2.8. Xây dựng lược đồ dịch trực tiếp cú pháp để dịch phát biểu FOR của ngôn ngữ C có dạng như sau: **FOR ( exp1; exp2; exp3 ) Stmt** sang dạng mà máy ảo kiểu Stack. Viết chương trình thực thi mã máy ảo kiểu Stack.

2.9. Xét đoạn văn phạm sau đây cho các câu lệnh if-then và if-then-else:

$$\begin{aligned} Stmt \rightarrow & \text{if expr then stmt} \\ & | \text{if expr then stmt else stmt} \\ & | \text{other} \end{aligned}$$

a) Chứng tỏ văn phạm này là văn phạm mơ hồ.

b) Xây dựng một văn phạm không mơ hồ tương đương với quy tắc: mỗi else chưa được kết hợp sẽ được kết hợp với then chưa kết hợp gần nhất trước đó.

c) Xây dựng một lược đồ dịch trực tiếp cú pháp để dịch các câu lệnh điều kiện thành mã máy ảo kiểu Stack.

2.10. Xây dựng lược đồ dịch trực tiếp cú pháp để dịch các phát biểu của ngôn ngữ PASCAL có dạng như sau sang dạng mà máy ảo kiểu Stack. Viết chương trình thực thi mã máy ảo kiểu Stack:

a) **REPEAT Stmt UNTIL expr**

b) **IF expr THEN Stmt ELSE Stmt**

c) **WHILE expr DO Stmt**

d) **FOR i := expr1 downto expr2 DO Stmt**

## CHƯƠNG III

### PHÂN TÍCH TỪ VỰNG

#### **Nội dung chính:**

Chương này trình bày các kỹ thuật xác định và cài đặt *bộ phân tích từ vựng*. Kỹ thuật đơn giản để xây dựng một bộ phân tích từ vựng là xây dựng các *lược đồ* - automata hữu hạn xác định (Deterministic Finite Automata - DFA) hoặc không xác định (Nondeterministic Finite Automata - NFA) – mô tả cấu trúc của các *thẻ từ* (token) của ngôn ngữ nguồn và sau đó dịch “thủ công” chúng sang chương trình nhận dạng các token. Một kỹ thuật khác nhằm tạo ra bộ phân tích từ vựng là sử dụng *Lex* – ngôn ngữ hành động theo *mẫu* (pattern). Trước tiên, người thiết kế trình biên dịch phải mô tả các mẫu được xác định bằng các biểu thức chính quy, sau đó sử dụng trình biên dịch của Lex để tự động tạo ra một bộ định dạng automata hữu hạn hiệu quả (bộ phân tích từ vựng). Các mô tả và cách thức hoạt động chi tiết của công cụ Lex được trình bày rõ hơn trong phần phụ lục A.

#### **Mục tiêu cần đạt:**

Sau khi học xong chương này, sinh viên phải nắm được các kỹ thuật tạo ra bộ phân tích từ vựng. Cụ thể,

- Xây dựng các lược đồ cho các biểu thức chính quy mô tả ngôn ngữ cần được viết trình biên dịch. Sau đó chuyển đổi chúng sang một chương trình phân tích từ vựng.
- Sử dụng công cụ có sẵn Lex để sinh ra bộ phân tích từ vựng.

#### **Kiến thức cơ bản:**

Sinh viên phải có các kiến thức về:

- DFA và NFA. Các automata hữu hạn xác định và không xác định này được sử dụng để nhận dạng chính xác ngôn ngữ mà các biểu thức chính quy có thể biểu diễn.
- Cách chuyển đổi từ NFA sang DFA nhằm làm đơn giản hóa quá trình cài đặt bộ phân tích từ vựng.

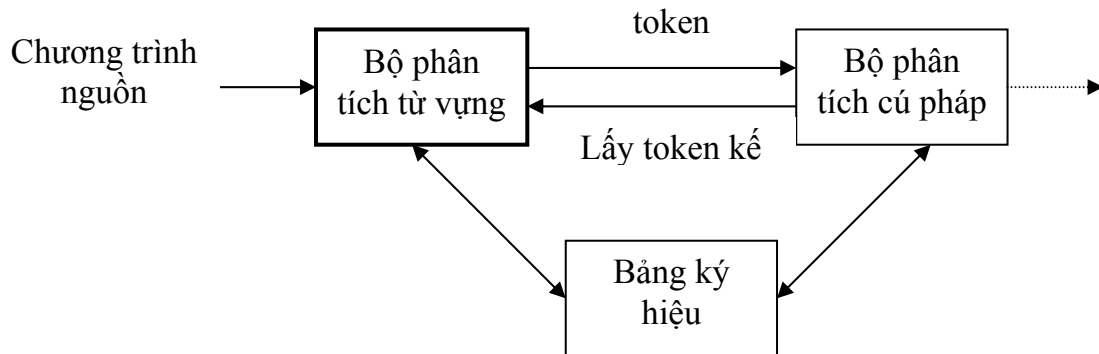
#### **Tài liệu tham khảo:**

- [1] **Automata and Formal Language. An Introduction** – Dean Kelley – Prentice Hall, Englewood Cliffs, New Jersey 07632.
- [2] **Compilers : Principles, Technique and Tools** - Alfred V.Aho, Jeffrey D.Ullman - Addison - Wesley Publishing Company, 1986.
- [3] **Compiler Design** – Reinhard Wilhelm, Dieter Maurer - Addison - Wesley Publishing Company, 1996.
- [4] **Design of Compilers : Techniques of Programming Language Translation** - Karen A. Lemone - CRC Press, Inc, 1992.
- [5] **Modern Compiler Implementation in C** - Andrew W. Appel - Cambridge University Press, 1997.



## I. VAI TRÒ CỦA BỘ PHÂN TÍCH TỪ VỰNG

Phân tích từ vựng là giai đoạn đầu tiên của mọi trình biên dịch. Nhiệm vụ chủ yếu của nó là đọc các ký hiệu nhập rồi tạo ra một chuỗi các token được sử dụng bởi bộ phân tích cú pháp. Sự tương tác này được thể hiện như hình sau, trong đó bộ phân tích từ vựng được thiết kế như một thủ tục được gọi bởi bộ phân tích cú pháp, trả về một token khi được gọi.



**Hình 3.1 - Giao diện của bộ phân tích từ vựng**

### 1. Các vấn đề của giai đoạn phân tích từ vựng

Có nhiều lý do để tách riêng giai đoạn phân tích từ vựng với giai đoạn phân tích cú pháp:

1. Thứ nhất, nó làm cho việc thiết kế đơn giản và dễ hiểu hơn. Chẳng hạn, bộ phân tích cú pháp sẽ không phải xử lý các khoảng trắng hay các lời chú thích nữa vì chúng đã được bộ phân tích từ vựng loại bỏ.

2. Hiệu quả của trình biên dịch cũng sẽ được cải thiện, nhờ vào một số chương trình xử lý chuyên dụng sẽ làm giảm đáng kể thời gian đọc dữ liệu từ chương trình nguồn và nhóm các token.

3. Tính đa tương thích (mang đi dễ dàng) của trình biên dịch cũng được cải thiện. Đặc tính của bộ ký tự nhập và những khác biệt của từng loại thiết bị có thể được giới hạn trong bước phân tích từ vựng. Dạng biểu diễn của các ký hiệu đặc biệt hoặc là những ký hiệu không chuẩn, chẳng hạn như ký hiệu ( trong Pascal có thể được cô lập trong bộ phân tích từ vựng.

### 2. Token, mẫu từ vựng và trị từ vựng

Khi nói đến bộ phân tích từ vựng, ta sẽ sử dụng các thuật ngữ *từ tố* (thẻ từ, token), *mẫu từ vựng* (pattern) và *trị từ vựng* (lexeme) với nghĩa cụ thể như sau:

- Từ tố (token) là các ký hiệu kết thúc trong văn phạm đối với một ngôn ngữ nguồn, chẳng hạn như: từ khóa, danh biểu, toán tử, dấu câu, hằng, chuỗi, ...
- Trị từ vựng (lexeme) của một token là một chuỗi ký tự biểu diễn cho token đó.
- Mẫu từ vựng (pattern) là qui luật mô tả một tập các trị từ vựng kết hợp với một token nào đó.

Một số ví dụ về cách dùng của các thuật ngữ này được trình bày trong bảng sau:

Token	Trị từ vựng minh họa	Mô tả của mẫu từ vựng
<b>const</b>	const	const
<b>if</b>	if	if
<b>relation</b>	<, <=, =, < >, >, >=	< hoặc <= hoặc = hoặc < > hoặc > hoặc >=
<b>id</b>	pi, count, d2	Mở đầu là chữ cái theo sau là chữ cái, chữ số
<b>num</b>	3.1416, 0, 5	Bất kỳ hằng số nào
<b>literal</b>	“ hello ”	Mọi chữ cái nằm giữa “ và “ ngoại trừ “

**Hình 3.2** - Các ví dụ về token

### 3. Thuộc tính của token

Khi có nhiều mẫu từ vựng khớp với một trị từ vựng, bộ phân tích từ vựng trong trường hợp này phải cung cấp thêm một số thông tin khác cho các bước biên dịch sau đó. Do đó đối với mỗi token, bộ phân tích từ vựng sẽ đưa thông tin về các token vào các thuộc tính đi kèm của chúng. Các token có ảnh hưởng đến các quyết định phân tích cú pháp; các thuộc tính ảnh hưởng đến việc phiên dịch các thẻ từ. Token kết hợp với thuộc tính của nó tạo thành một bộ *<token, tokenval>*.

**Ví dụ 3.1:** Token và giá trị thuộc tính đi kèm của câu lệnh Fortran :  $E = M * C ** 2$  được viết như một dãy các bộ sau:

< **id**, con trỏ trong bảng ký hiệu của E >  
 < **assign\_op**,     >  
 < **id**, con trỏ trong bảng ký hiệu của M >  
 < **mult\_op**,     >  
 < **id**, con trỏ trong bảng ký hiệu của C >  
 < **exp\_op**,     >  
 < **num**, giá trị nguyên 2 >

Chú ý rằng một số bộ không cần giá trị thuộc tính, thành phần đầu tiên là đủ để nhận dạng trị từ vựng.

### 4. Lỗi từ vựng

Chỉ một số ít lỗi được phát hiện tại bước phân tích từ vựng, bởi vì bộ phân tích từ vựng có nhiều cách nhìn nhận chương trình nguồn. Ví dụ chuỗi **fi** được nhìn thấy lần đầu tiên trong một chương trình C với ngữ cảnh : **fi ( a == f (x)) ...** Bộ phân tích từ vựng không thể biết đây là lỗi không viết đúng từ khóa **if** hay một danh biểu chưa được khai báo. Vì **fi** là một danh biểu hợp lệ nên bộ phân tích từ vựng phải trả về một token và để một giai đoạn khác sau đó xác định lỗi. Tuy nhiên, trong một vài tình huống phải khắc phục lỗi để phân tích tiếp. Chiến lược đơn giản nhất là "*phương thức hoảng sợ*" (panic mode): Các ký tự tiếp theo sẽ được xóa ra khỏi chuỗi nhập còn lại

cho đến khi tìm ra một token hoàn chỉnh. Kỹ thuật này đôi khi cũng gây ra sự nhầm lẫn cho giai đoạn phân tích cú pháp, nhưng nói chung là vẫn có thể sử dụng được.

Một số chiến lược khắc phục lỗi khác là:

1. Xóa đi một ký tự dư.
2. Xen thêm một ký tự bị mất.
3. Thay thế một ký tự không đúng bằng một ký tự đúng.
4. Chuyển đổi hai ký tự kế tiếp nhau.

## II. LƯU TRỮ TẠM CHƯƠNG TRÌNH NGUỒN

Việc đọc từng ký tự trong chương trình nguồn có thể tiêu hao một số thời gian đáng kể do đó ảnh hưởng đến tốc độ dịch. Để giải quyết vấn đề này người ta đọc một lúc một chuỗi ký tự, lưu trữ vào trong vùng nhớ tạm - gọi là *bộ đệm input* (buffer). Tuy nhiên, việc đọc như vậy cũng gặp một số trở ngại do không thể xác định một chuỗi như thế nào thì chứa trọn vẹn một token? Phần này giới thiệu vài phương pháp đọc bộ đệm hiệu quả:

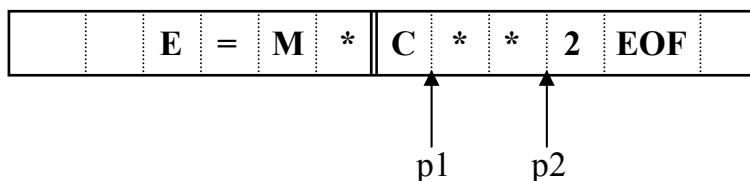
### 1. Cặp bộ đệm (Buffer Pairs)

Đối với nhiều ngôn ngữ nguồn, có một vài trường hợp bộ phân tích từ vựng phải đọc thêm một số ký tự trong chương trình nguồn vượt quá trị từ vựng cho một mẫu trước khi có thể thông báo đã so trùng được một token.

Trong phương pháp cặp bộ đệm, vùng đệm sẽ được chia thành hai nửa với kích thước bằng nhau, mỗi nửa chứa được N ký tự. Thông thường, N là số ký tự trên một khối đĩa, N bằng 1024 hoặc 4096.

Mỗi lần đọc, N ký tự từ chương trình nguồn sẽ được đọc vào mỗi nửa bộ đệm bằng một lệnh đọc (read) của hệ thống. Nếu số ký tự còn lại trong chương trình nguồn ít hơn N thì một ký tự đặc biệt eof được đưa vào buffer sau các ký tự vừa đọc để báo hiệu chương trình nguồn đã được đọc hết.

Sử dụng hai con trỏ dò tìm trong buffer. Chuỗi ký tự nằm giữa hai con trỏ luôn luôn là trị từ vựng hiện hành. Khởi đầu, cả hai con trỏ đặt trùng nhau tại vị trí bắt đầu của mỗi trị từ vựng. Con trỏ p1 (lexeme\_beginning) - con trỏ bắt đầu trị từ vựng - sẽ giữ cố định tại vị trí này cho đến khi con trỏ p2 (forwar) - con trỏ tới - di chuyển qua từng ký tự trong buffer để xác định một token. Khi một trị từ vựng cho một token đã được xác định, con trỏ p1 dời lên trùng với p2 và bắt đầu dò tìm một trị từ vựng mới.



**Hình 3.3** - Cặp hai nửa vùng đệm

Khi con trỏ p2 tới ranh giới giữa 2 vùng đệm, nửa bên phải được lấp đầy bởi N ký tự tiếp theo trong chương trình nguồn. Khi con trỏ p2 tới vị trí cuối bộ đệm, nửa bên trái sẽ được lấp đầy bởi N ký tự mới và p2 sẽ được dời về vị trí bắt đầu bộ đệm.

Phương pháp cặp bộ đệm này thường hoạt động rất tốt nhưng khi đó số lượng ký tự đọc trước bị giới hạn và trong một số trường hợp nó có thể không nhận dạng được token khi con trỏ p2 phải vượt qua một khoảng cách lớn hơn chiều dài vùng đệm.

Giải thuật hình thức cho hoạt động của con trỏ p2 trong bộ đệm :

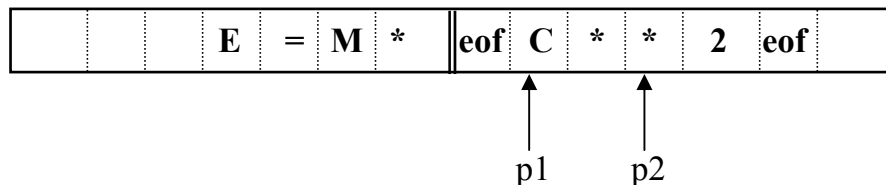
```

if p2 ở cuối nửa đầu then
    begin
        Đọc vào nửa cuối;
        p2 := p2 + 1;
    end
else if p2 ở cuối của nửa cuối then
    begin
        Đọc vào nửa đầu;
        Dời p2 về đầu bộ đệm ;
    end
else p2 := p2 + 1

```

## 2. Khóa cảm canh (Sentinel)

Phương pháp cặp bộ đệm đòi hỏi mỗi lần di chuyển p2 đều phải kiểm tra xem có phải đã hết một nửa buffer chưa nên kém hiệu quả vì phải hai lần kiểm tra. Để khắc phục điều này, mỗi lần chỉ đọc N-1 ký tự vào mỗi nửa buffer còn ký tự thứ N là một ký tự đặc biệt, thường là eof. Như vậy chúng ta đã rút ngắn một lần kiểm tra.



**Hình 3.4 - Khóa cảm canh eof tại cuối mỗi vùng đệm**

Giải thuật hình thức cho hoạt động của con trỏ p2 trong bộ đệm :

```

p2 := p2 + 1;
if p2↑ = eof then
    begin
        if p2 ở cuối của nửa đầu then
            begin
                Đọc vào nửa cuối;
                p2 := p2 + 1;
            end
        else if p2 ở cuối của nửa sau then

```

```

begin
    Đọc vào nửa đầu;
    Dời p2 vào đầu của nửa đầu;
end
else      /* EOF ở giữa vùng đệm chỉ hết chương trình nguồn */
    kết thúc phân tích từ vựng;
end

```

### III. ĐẶC TẢ TOKEN (Specification of Token )

#### 1. Chuỗi và ngôn ngữ

Chuỗi là một tập hợp hữu hạn các ký tự. Độ dài chuỗi là số các ký tự trong chuỗi. Chuỗi rỗng  $\epsilon$  là chuỗi có độ dài 0.

Ngôn ngữ là tập hợp các chuỗi. Ngôn ngữ có thể chỉ bao gồm một chuỗi rỗng ký hiệu là  $\emptyset$ .

#### 2. Các phép toán trên ngôn ngữ

Cho 2 ngôn ngữ L và M :

- **Hợp** của L và M :  $L \cup M = \{ s \mid s \in L \text{ hoặc } s \in M \}$
- **Ghép** (concatenation) của L và M:  $LM = \{ st \mid s \in L \text{ và } t \in M \}$
- **Bao đóng Kleen** của L:  $L^* = \bigcup_{i=0}^{\infty} L^i$   
(Ghép của 0 hoặc nhiều L)
- **Bao đóng dương** (positive closure) của L:  $L^+ = \bigcup_{i=1}^{\infty} L^i$   
(Ghép của 1 hoặc nhiều L)

**Ví dụ 3.2:**  $L = \{A, B, \dots, Z, a, b, \dots, z\}$

$D = \{0, 1, \dots, 9\}$

1.  $L \cup D$  là tập hợp các chữ cái và số.
2.  $LD$  là tập hợp các chuỗi bao gồm một chữ cái và một chữ số.
3.  $L^4$  là tập hợp tất cả các chuỗi 4 chữ cái.
4.  $L^*$  là tập hợp tất cả các chuỗi của các chữ cái bao gồm cả chuỗi rỗng.
5.  $L(L \cup D)^*$  là tập hợp tất cả các chuỗi mở đầu bằng một chữ cái theo sau là chữ cái hay chữ số
6.  $D^+$  là tập hợp tất cả các chuỗi gồm một hoặc nhiều chữ số.

#### 3. Biểu thức chính quy (Regular Expression)

Trong Pascal, một danh biểu là một phần tử của tập hợp  $L(L \cup D)^*$ . Chúng ta có thể viết: `danhbiểu = letter (letter | digit)*` - Đây là một biểu thức chính quy.

**Biểu thức chính quy** được xây dựng trên một tập hợp các luật xác định. Mỗi biểu thức chính quy  $r$  đặc tả một ngôn ngữ  $L(r)$ .

Sau đây là các luật xác định biểu thức chính quy trên tập Alphabet  $\Sigma$ .

1.  $\varepsilon$  là một biểu thức chính quy đặc tả cho một chuỗi rỗng  $\{\varepsilon\}$ .
2. Nếu  $a \in \Sigma$  thì  $a$  là biểu thức chính quy  $r$  đặc tả tập hợp các chuỗi  $\{a\}$
3. Giả sử  $r$  và  $s$  là các biểu thức chính quy đặc tả các ngôn ngữ  $L(r)$  và  $L(s)$  ta có:
  - a.  $(r) | (s)$  là một biểu thức chính quy đặc tả  $L(r) \cup L(s)$
  - b.  $(r)(s)$  là một biểu thức chính quy đặc tả  $L(r)L(s)$ .
  - c.  $(r)^*$  là một biểu thức chính quy đặc tả  $(L(r))^*$

*Quy ước:*

Toán tử bao đóng  $*$  có độ ưu tiên cao nhất và kết hợp trái.

Toán tử ghép có độ ưu tiên thứ hai và kết hợp trái.

Toán tử hợp  $|$  có độ ưu tiên thấp nhất và kết hợp trái.

**Ví dụ 3.3:** Cho  $\Sigma = \{a, b\}$

1. Biểu thức chính quy  $a | b$  đặc tả  $\{a, b\}$
2. Biểu thức chính quy  $(a | b)(a | b)$  đặc tả tập hợp  $\{aa, ab, ba, bb\}$ . Tập hợp này có thể được đặc tả bởi biểu thức chính quy tương đương sau:  $aa | ab | ba | bb$ .
3. Biểu thức chính quy  $a^*$  đặc tả  $\{\varepsilon, a, aa, aaa, \dots\}$
4. Biểu thức chính quy  $(a | b)^*$  đặc tả  $\{\varepsilon, a, b, aa, bb, \dots\}$ . Tập này có thể đặc tả bởi  $(a^*b^*)^*$ .
5. Biểu thức chính quy  $a | a^*b$  đặc tả  $\{a, b, ab, aab, \dots\}$

Hai biểu thức chính quy cùng đặc tả một tập hợp ta nói rằng chúng tương đương và viết  $r = s$ .

#### 4. Các tính chất đại số của biểu thức chính quy

Biểu thức chính quy cũng tuân theo một số luật đại số và có thể dùng các luật này để biến đổi biểu thức thành những dạng tương đương. Bảng sau trình bày một số luật đại số cho các biểu thức chính quy  $r, s$  và  $t$ .

Tính chất	Mô tả
$r   s = s   r$	$ $ có tính chất giao hoán
$r   (s   t) = (r   s)   t$	$ $ có tính chất kết hợp
$(rs)t = r(st)$	Phép ghép có tính chất kết hợp
$r(s   t) = rs   rt$ $(s   t)r = sr   tr$	Phép ghép phân phối đối với phép $ $
$\varepsilon r = r$	$\varepsilon$ là phần tử đơn vị của phép ghép

$r\varepsilon = r$	
$r^* = (r \mid \varepsilon)^*$	Quan hệ giữa $r$ và $\varepsilon$
$r^{**} = r^*$	$*$ có hiệu lực như nhau

**Hình 3.5** - Một số tính chất đại số của biểu thức chính quy

## 5. Định nghĩa chính quy (Regular Definitions)

Định nghĩa chính quy là một chuỗi các định nghĩa có dạng :

$d_1 \rightarrow r_1$   $d_i$  là một tên,  
 $d_2 \rightarrow r_2$   $r_i$  là một biểu thức chính quy.  
 ...  
 $d_n \rightarrow r_n$

**Ví dụ 3.4:** Tập hợp các danh biểu trong Pascal là một tập hợp các chuỗi chữ cái và số, mở đầu bằng một chữ cái. Định nghĩa chính quy của tập đó là:

$\text{letter} \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$   
 $\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$   
 $\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$

**Ví dụ 3.5 :** Các số không dấu trong Pascal là các chuỗi 5280, 39.37, 6.336E4 hoặc 1.894E-4. Định nghĩa chính quy sau đặc tả tập các số này là :

$\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$   
 $\text{digits} \rightarrow \text{digit} \text{ digit}^*$   
 $\text{optional\_fraction} \rightarrow . \text{digits} \mid \varepsilon$   
 $\text{optional\_exponent} \rightarrow ( E ( + \mid - \mid \varepsilon ) \text{digits} ) \mid \varepsilon$   
 $\text{num} \rightarrow \text{digits optional\_fraction optional\_exponent}$

## 6. Ký hiệu viết tắt

Người ta quy định các ký hiệu viết tắt cho thuận tiện trong việc biểu diễn như sau:

1. Một hoặc nhiều: dùng dấu +
2. Không hoặc một: dùng dấu ?

**Ví dụ 3.6:**  $r \mid \varepsilon$  được viết tắt là  $r?$

**Ví dụ 3.7:** Viết tắt cho định nghĩa chính quy tập hợp số **num** trong ví dụ 3.5

$\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$   
 $\text{digits} \rightarrow \text{digit}^+$   
 $\text{optional\_fraction} \rightarrow ( . \text{digits} ) ?$   
 $\text{optional\_exponent} \rightarrow ( E ( + \mid - ) ? \text{digits} ) ?$   
 $\text{num} \rightarrow \text{digits optional\_fraction optional\_exponent}$

### 3. Lớp ký tự

$[abc] = a \mid b \mid c$

$[a - z] = a \mid b \mid \dots \mid z$

Sử dụng lớp ký hiệu chúng ta có thể mô tả danh biểu như là một chuỗi sinh ra bởi biểu thức chính quy :

$[A - Z a - z] [A - Z a - z 0 - 9]^*$

## IV. NHẬN DẠNG TOKEN

Trong suốt phần này, chúng ta sẽ dùng ngôn ngữ được tạo ra bởi văn phạm dưới đây làm thí dụ minh họa :

$\text{stmt} \rightarrow \text{if expr then stmt}$   
 $\quad \mid \text{if expr then stmt else stmt}$   
 $\quad \mid \epsilon$   
 $\text{expr} \rightarrow \text{term relop term}$   
 $\quad \mid \text{term}$   
 $\text{term} \rightarrow \text{id}$   
 $\quad \mid \text{num}$

Trong đó các ký hiệu kết thúc if, then, else, relop, id, num được cho bởi định nghĩa chính quy sau:

$\text{if} \rightarrow \text{if}$   
 $\text{then} \rightarrow \text{then}$   
 $\text{else} \rightarrow \text{else}$   
 $\text{relop} \rightarrow < \mid <= \mid = \mid <> \mid > \mid >=$   
 $\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$   
 $\text{num} \rightarrow \text{digit}^+ ( . \text{digit}^+ ) ? ( \text{E} ( + \mid - ) ? \text{digit}^+ ) ?$

Định nghĩa chính quy của các khoảng trắng ws (white space)

$\text{delim} \rightarrow \text{blank} \mid \text{tab} \mid \text{newline}$   
 $\text{ws} \rightarrow \text{delim}^+$

Mục đích của chúng ta là xây dựng một bộ phân tích từ vựng có thể định vị được từ tổ cho các token kế tiếp trong vùng đệm và tạo ra output là một cặp token thích hợp và giá trị thuộc tính của nó bằng cách dùng mẫu biểu thức chính quy cho các token như sau:

Biểu thức chính quy	Token	Trị thuộc tính
<b>ws</b>	-	-
<b>if</b>	<b>if</b>	-
<b>then</b>	<b>then</b>	-



<b>else</b>	<b>else</b>	-
<b>id</b>	<b>id</b>	con trỏ trong bảng ký hiệu
<b>num</b>	<b>num</b>	giá trị số
<	<b>relop</b>	LT (Less Than)
<=	<b>relop</b>	LE (Less Or Equal)
=	<b>relop</b>	EQ (Equal)
<>	<b>relop</b>	NE (Not Equal)
>	<b>relop</b>	GT (Greater Than)
>=	<b>relop</b>	GE (Greater Or Equal)

**Hình 3.6** - Mẫu biểu thức chính quy cho một số token

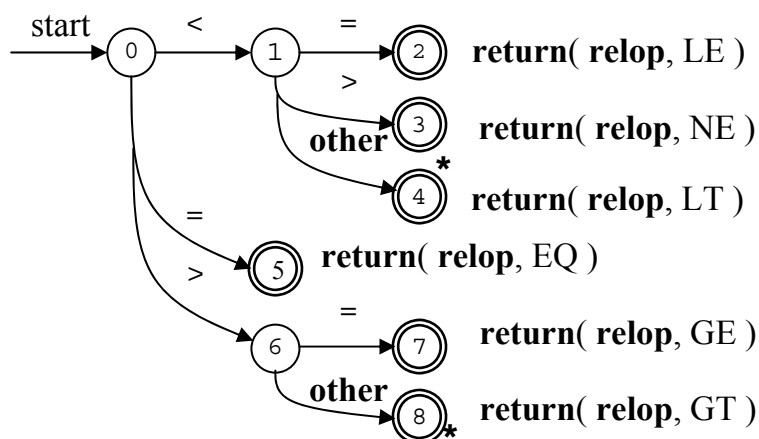
### 1. Sơ đồ dịch

Để dễ dàng nhận dạng token, chúng ta xây dựng cho mỗi token một *sơ đồ dịch* (translation diagram). Sơ đồ dịch bao gồm các trạng thái (state) ký hiệu bởi vòng tròn và các cạnh mũi tên nối các trạng thái.

Nói chung thường có nhiều sơ đồ dịch, mỗi sơ đồ đặc tả một nhóm token. Nếu xảy ra thất bại khi chúng ta đang đi theo một sơ đồ dịch thì chúng ta dịch lui con trỏ tới về nơi nó đã ở trong trạng thái khởi đầu của sơ đồ này rồi kích hoạt sơ đồ dịch tiếp theo. Do con trỏ đầu từ vựng và con trỏ tới cùng chỉ đến một vị trí trong trạng thái khởi đầu của sơ đồ, con trỏ tới sẽ được dịch lui lại để chỉ đến vị trí được con trỏ đầu từ vựng chỉ tới. Nếu xảy ra thất bại trong tất cả mọi sơ đồ dịch thì xem như một lỗi từ vựng đã được phát hiện và chúng ta sẽ khởi động một thủ tục khắc phục lỗi.

Phần dưới đây trình bày một số sơ đồ dịch nhận dạng các token trong văn phạm ví dụ trên.

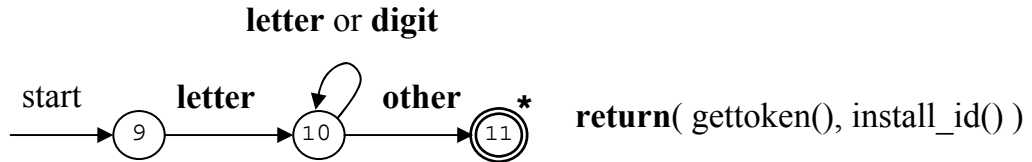
Sơ đồ dịch nhận dạng cho token **relop**:



**Hình 3.7** - Sơ đồ dịch cho các toán tử quan hệ

Chúng ta dùng ký hiệu \* để chỉ ra những trạng thái mà chúng ta đã đọc quá một ký tự, cần phải quay lui con trỏ lại.

Sơ đồ dịch nhận dạng token **id**:

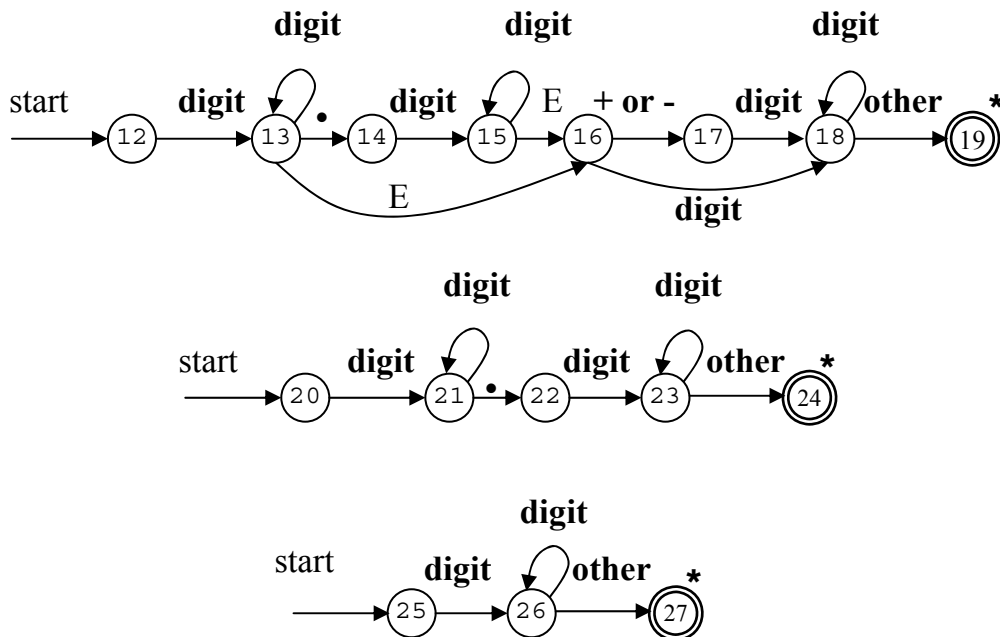


**Hình 3.8** - Sơ đồ dịch cho các danh biểu và từ khóa

Một kỹ thuật đơn giản để tách từ khóa ra khỏi các danh biểu là khởi tạo bảng ký hiệu lưu trữ thông tin về danh biểu một cách thích hợp. Đối với các token cần nhận dạng trong văn phạm này, chúng ta cần nhập các chuỗi **if**, **then** và **else** vào bảng ký hiệu trước khi đọc các ký hiệu trong bộ đệm nguyên liệu. Đồng thời ghi chú trong bảng ký hiệu để trả về token đó khi một trong các chuỗi này được nhận ra. Sử dụng các hàm `gettoken()` và `install_id()` tương ứng để nhận token và các thuộc tính trả về.

Sơ đồ dịch nhận dạng token **num**:

Một số vấn đề sẽ nảy sinh khi chúng ta xây dựng bộ nhận dạng cho các số không dấu. Trị từ vựng cho một token num phải là trị từ vựng dài nhất có thể được. Do đó, việc thử nhận dạng số trên các sơ đồ dịch phải theo thứ tự từ sơ đồ nhận dạng số dài nhất.



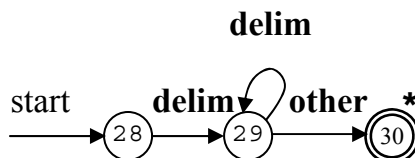
**Hình 3.9** - Sơ đồ dịch cho các số không dấu trong Pascal

Có nhiều cách để tránh các đối sánh dư thừa trong các sơ đồ dịch trên. Một cách là viết lại các sơ đồ dịch bằng cách tổ hợp chúng thành một - một công việc nói chung là không đơn giản lắm. Một cách khác là thay đổi cách đáp ứng với thất bại trong quá trình duyệt qua một sơ đồ. Phương pháp được sử dụng ở đây là cho phép ta vượt qua nhiều trạng thái kiểm nhận và quay trở lại trạng thái kiểm nhận cuối cùng đã đi qua khi thất bại xảy ra.

Sơ đồ dịch nhận dạng khoảng trắng **ws (white space)**:

Việc xử lý các khoảng trắng ws không hoàn toàn giống như các mẫu nói trên bởi vì không có gì để trả về cho bộ phân tích cú pháp khi tìm thấy các khoảng trắng trong

chuỗi nhập. Và do đó, thao tác đơn giản cho việc dò tìm trên sơ đồ dịch khi phát hiện khoảng trắng là trở lại trạng thái bắt đầu của sơ đồ dịch đầu tiên để tìm một mẫu khác.



**Hình 3.10** - Sơ đồ dịch cho các khoảng trắng

## 2. Cài đặt một sơ đồ dịch

Dãy các sơ đồ dịch có thể được chuyển thành một chương trình để tìm kiếm token được đặc tả bằng các sơ đồ. Mỗi trạng thái tương ứng với một đoạn mã chương trình. Nếu có các cạnh đi ra từ trạng thái thì đọc một ký tự và tùy thuộc vào ký tự đó mà đi đến trạng thái khác. Ta dùng hàm **nextchar()** đọc một ký tự từ trong bộ đệm input và con trỏ p2 di chuyển sang phải một ký tự. Nếu không có một cạnh đi ra từ trạng thái hiện hành phù hợp với ký tự vừa đọc thì con trỏ p2 phải quay lại vị trí của p1 để chuyển sang sơ đồ dịch kế tiếp. Hàm **fail()** sẽ làm nhiệm vụ này. Nếu không có sơ đồ nào khác để thử, **fail()** sẽ gọi một thủ tục khắc phục lỗi.

Để trả về các token, chúng ta dùng một biến toàn cục **lexical\_value**. Nó được gán cho các con trỏ được các hàm **install\_id()** và **install\_num()** trả về, tương ứng khi tìm ra một danh biểu hoặc một số. Lóp token được trả về bởi thủ tục chính của bộ phân tích từ vựng có tên là **nexttoken()**.

```
int state = 0, start = 0;
int lexical_value;      /* để “trả về” thành phần thứ hai của token */
```

```
int fail ( )
{
    forward = token_beginning;
    switch (start) {
        case 0 : start = 9; break;
        case 9 : start = 12; break;
        case 12 : start = 20; break;
        case 20 : start = 25; break;
        case 25 : recover ( ); break;
        default : /* lỗi trình biên dịch */
    }
    return start;
}
```

```
token    nexttoken ( )
```

```

{ while (1) {
    switch (state) {
    case 0 : c = nextchar ( ) ;    /* c là ký hiệu đọc trước */
        if ( c == blank || c == tab || c == newline ) {
            state = 0;
            lexeme_beginning ++ ;    /* dịch con trỏ đến đầu từ vựng */
        }
        else if (c == ' < ' ) state = 1;
        else if (c == ' = ' ) state = 5;
        else if (c == ' > ' ) state = 6;
        else state = fail ( ) ; break ;

        ...    /* các trường hợp 1- 8 ở đây */

[ case 9 : c = nextchar ( ) ;
    if (isletter (c)) state=10;
    else state = fail ( ) ; break ;
case 10 : c = nextchar ( ) ;
    if (isletter (c)) state=10;
    else if (isdigit(c)) state = 10 ;
        else state = 11 ; break ;
case 11 : retract (1) ; install_id ( ) ;
        return (gettoken ( ));

        ...    /* các trường hợp 12 - 24 ở đây */

case 25 : c = nextchar ( ) ;
    if (isdigit (c)) state=26;
    else state = fail ( ) ; break ;
case 26 : c = nextchar ( ) ;
    if (isdigit (c)) state=26;
    else state = 27 ; break ;
case 27 : retract (1) ; install_num ( ) ;
        return (NUM);

```

```

    }
}
}

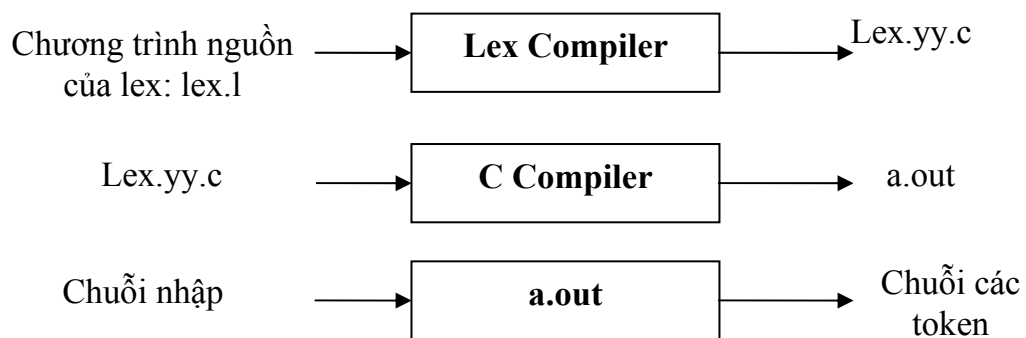
```

## V. NGÔN NGỮ ĐẶC TẢ CHO BỘ PHÂN TÍCH TỪ VỰNG

### 1. Bộ sinh bộ phân tích từ vựng

Có nhiều công cụ để xây dựng bộ phân tích từ vựng dựa vào các biểu thức chính quy. **Lex** là một công cụ được sử dụng rộng rãi để tạo bộ phân tích từ vựng.

Trước hết đặc tả cho một bộ phân tích từ vựng được chuẩn bị bằng cách tạo ra một chương trình **lex.l** trong ngôn ngữ lex. Trình biên dịch Lex sẽ dịch lex.l thành một chương trình C là **lex.yy.c**. Chương trình này bao gồm các đặc tả về sơ đồ dịch được xây dựng từ các biểu thức chính quy của **lex.l**, kết hợp với các thủ tục chuẩn nhận dạng từ vựng. Các hành vi kết hợp với biểu thức chính quy trong **lex.l** là các đoạn chương trình C được chuyển sang **lex.yy.c**. Cuối cùng trình biên dịch C sẽ dịch lex.yy.c thành chương trình đối tượng **a.out**, đó là bộ phân tích từ vựng có thể chuyển dòng nhập thành chuỗi các token.



**Hình 3.11** - Tạo ra bộ phân tích từ vựng bằng Lex

**Chú ý:** Những điều ta nói trên là nói về lex trong UNIX. Ngày nay có nhiều version của lex như Lex cho Pascal hoặc Javalex.

### 2. Đặc tả lex

Một chương trình lex bao gồm 3 thành phần:

#### Khai báo

%%

#### Quy tắc dịch

%%

#### Các thủ tục phụ

**Phần khai báo** bao gồm khai báo biến, hằng và các định nghĩa chính quy.

**Phần quy tắc dịch** cho các lệnh có dạng:

```
p1 {action 1 }
```

```

p2      {action 2 }
. . .
pn      {action n }

```

Trong đó  $p_i$  là các biểu thức chính quy, action  $i$  là đoạn chương trình mô tả hành động của bộ phân tích từ vựng thực hiện khi  $p_i$  tương ứng phù hợp với từ vựng. Trong lex các đoạn chương trình này được viết bằng C nhưng nói chung có thể viết bằng bất cứ ngôn ngữ nào.

**Các thủ tục phụ** là sự cài đặt các hành động trong phần 2.

**Ví dụ 3.8:** Sau đây trình bày một chương trình Lex nhận dạng các token của văn phạm đã nêu ở phần trước và trả về token được tìm thấy.

```

%{
/* định nghĩa các hằng
LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUMBER, RELOP */
}%
/* định nghĩa chính quy */
delim      [\t\n]
ws          {delim}+
letter     [A - Za - z]
digit      [0 - 9]
id          {letter}({letter}| {digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}       {/* Không có action, không có return */}
if          {return(IF); }
then        {return(THEN); }
else        {return(ELSE); }
{id}        {yyval = install_id( ); return(ID) }
{number}    {yyval = install_num( ); return(NUMBER) }
"<"        {yyval = LT; return(RELOP) }
"<="       {yyval = LE; return(RELOP) }
"="         {yyval = EQ; return(RELOP) }
"<>"        {yyval = NE; return(RELOP) }
">"         {yyval = GT; return(RELOP) }
">="       {yyval = GE; return(RELOP) }
%%

```

```
install_id ( ) {  
    /* Thủ tục phụ cài id vào trong bảng ký hiệu */  
}  
  
install_num ( ) {  
    /* Thủ tục phụ cài một số vào trong bảng ký hiệu */  
}
```

## BÀI TẬP CHƯƠNG III

**3.1.** Xác định bộ chữ cái của các ngôn ngữ sau:

- a) Pascal
- b) C
- c) LISP

**3.2.** Hãy xác định các từ vựng có thể hình thành các token trong các đoạn chương trình sau:

a) **PASCAL**

```
function max (i, j :integer) : integer;  
{ Trả về số nguyên lớn hơn trong 2 số i và j }  
begin  
    i > j then max := i  
    else max := j;  
end;
```

b) **C**

```
int max (i, j) int i, j;    /* Trả về số nguyên lớn hơn trong 2 số i và j */  
{ return i > j ? i : j  
}
```

c) **FORTRAN 77**

```
FUNCTION MAX (i, j)  
C  Trả về số nguyên lớn hơn trong 2 số i và j  
IF ( I.GT. J) THEN  
    MAX = I  
ELSE  
    MAX = J  
END IF  
RETURN
```



**3.3.** Viết một chương trình Lex sao chép một tập tin, thay các chuỗi khoảng trắng thành một khoảng trắng duy nhất.

**3.4.** Viết một đặc tả Lex cho các token của ngôn ngữ Pascal và dùng trình biên dịch Lex để xây dựng một bộ phân tích từ vựng cho Pascal.

## CHƯƠNG IV

### PHÂN TÍCH CÚ PHÁP

#### Nội dung chính:

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

#### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải nắm được:

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

#### Kiến thức cơ bản:

Sinh viên phải có các kiến thức về:

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

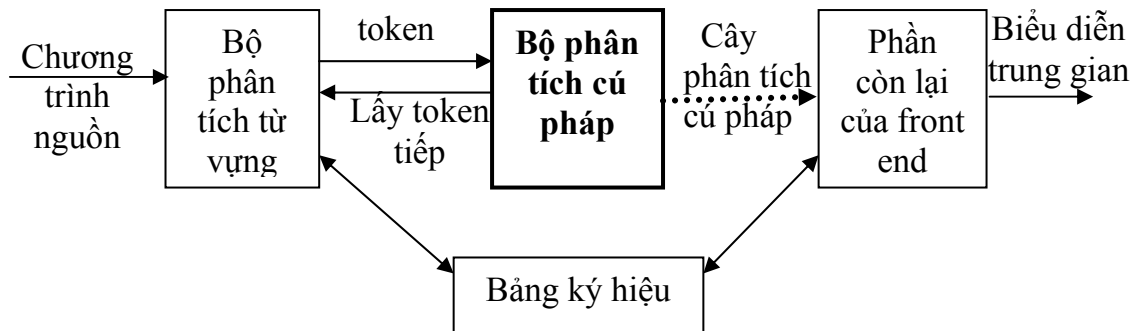
#### Tài liệu tham khảo:

- [1] **Automata and Formal Language. An Introduction** – Dean Kelley – Prentice Hall, Englewood Cliffs, New Jersey 07632.
- [2] **Compilers : Principles, Technique and Tools** - Alfred V.Aho, Jeffrey D.Ullman - Addison - Wesley Publishing Company, 1986.
- [3] **Compiler Design** – Reinhard Wilhelm, Dieter Maurer - Addison - Wesley Publishing Company, 1996.
- [4] **Design of Compilers : Techniques of Programming Language Translation** - Karen A. Lemone - CRC Press, Inc, 1992.
- [5] **Modern Compiler Implementation in C** - Andrew W. Appel - Cambridge University Press, 1997.

## I. VAI TRÒ CỦA BỘ PHÂN TÍCH CÚ PHÁP

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

Bộ phân tích cú pháp nhận chuỗi các token từ bộ phân tích từ vựng và xác nhận rằng chuỗi này có thể được sinh ra từ văn phạm của ngôn ngữ nguồn bằng cách tạo ra cây phân tích cú pháp cho chuỗi. Bộ phân tích cú pháp cũng có cơ chế ghi nhận các lỗi cú pháp theo một phương thức linh hoạt và có khả năng phục hồi được các lỗi thường gặp để có thể tiếp tục xử lý phần còn lại của chuỗi nhập.



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

### 2. Xử lý lỗi cú pháp

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

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

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

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

### 3. Các chiến lược phục hồi lỗi

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

**a. Phương thức "hoảng sợ" (panic mode recovery):** Đây là phương pháp đơn giản nhất cho cài đặt và có thể dùng cho hầu hết các phương pháp phân tích. Khi một

lỗi được phát hiện thì bộ phân tích cú pháp bỏ qua từng ký hiệu một cho đến khi tìm thấy một tập hợp được chỉ định của các token đồng bộ (synchronizing tokens), các token đồng bộ thường là dấu chấm phẩy (;) hoặc end.

**b. Chiến lược mức ngữ đoạn (phrase\_level recovery):** Khi phát hiện một lỗi, bộ phân tích cú pháp có thể thực hiện sự hiệu chỉnh cục bộ trên phần còn lại của dòng nhập. Cụ thể là thay thế phần đầu còn lại bằng một chuỗi ký tự có thể tiếp tục. Chẳng hạn, dấu phẩy (,) bởi dấu chấm phẩy (;), xóa một dấu phẩy lạ hoặc thêm vào một dấu chấm phẩy.

**c. Chiến lược dùng các luật sinh sửa lỗi (error production):** Thêm vào văn phạm của ngôn ngữ những luật sinh lỗi và sử dụng văn phạm này để xây dựng bộ phân tích cú pháp, chúng ta có thể sinh ra bộ đoán lỗi thích hợp để chỉ ra cấu trúc lỗi được nhận biết trong dòng nhập.

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

## II. BIẾN ĐỔI VĂN PHẠM PHI NGỮ CẢNH

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

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

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

$$\begin{aligned} E &\rightarrow E A E \mid (E) \mid -E \mid id \\ A &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

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

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

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

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

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

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

Ta có tính chất:

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

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

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

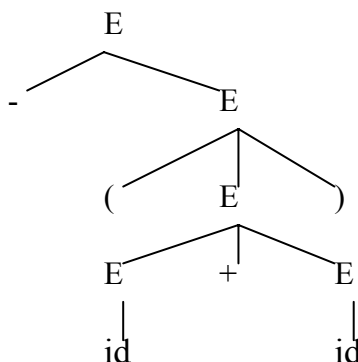
Một cây phân tích cú pháp có thể xem như một biểu diễn đồ thị cho một dẫn xuất.

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

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

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

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



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

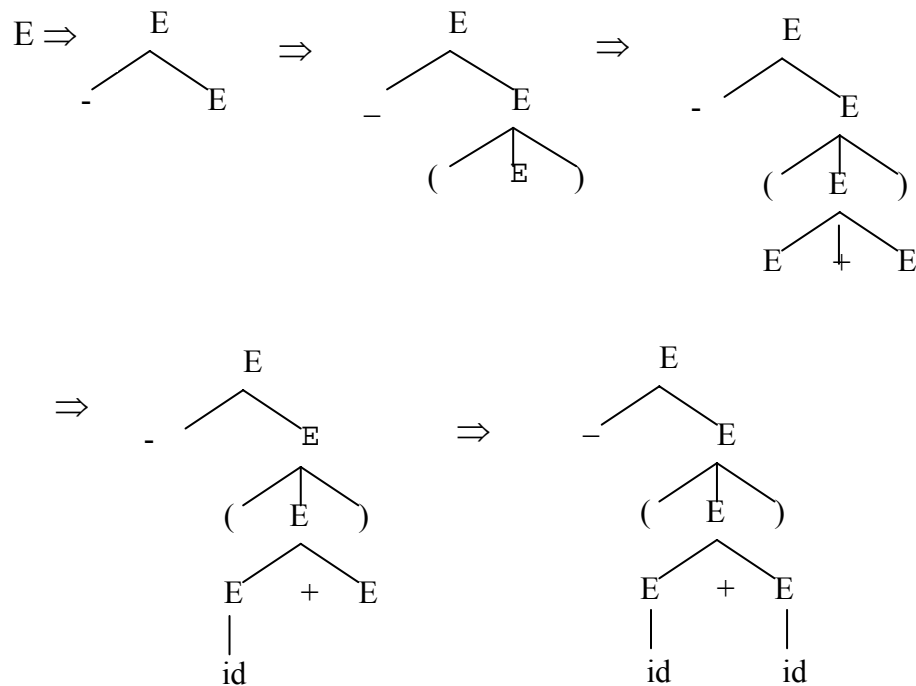
Để thấy mối quan hệ giữa cây phân tích cú pháp và dẫn xuất, ta xét một dẫn xuất :

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$  trong đó  $\alpha_i$  là một ký hiệu chưa kết thúc  $A$ .

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

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

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



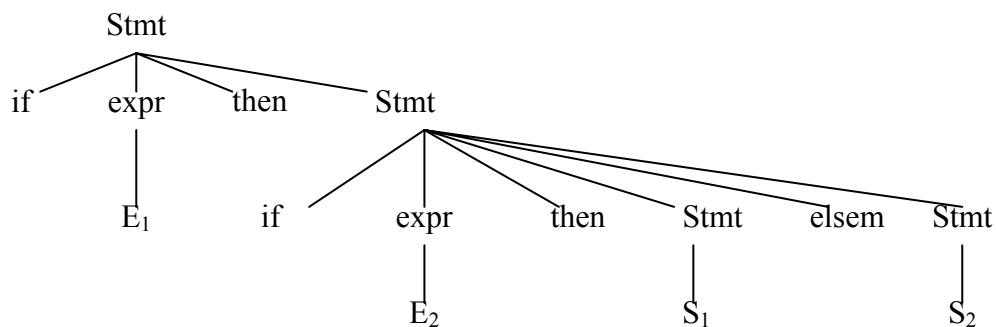
**Hình 4.3** - Xây dựng cây phân tích cú pháp từ dẫn xuất

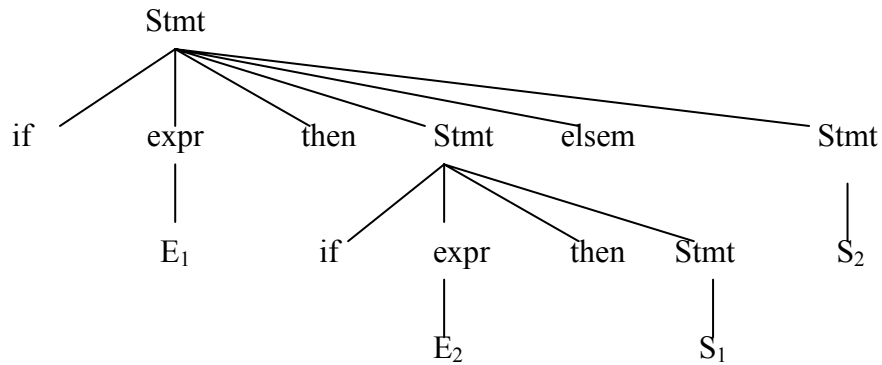
## 2. Loại bỏ sự mơ hồ

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

$\text{Stmt} \rightarrow$  **if** expr **then** stmt  
           | **if** expr **then** stmt **else** stmt  
           | **other**

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





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

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

$\text{Stmt} \rightarrow \text{matched\_stmt} \mid \text{unmatched\_stmt}$   
 $\text{matched\_stmt} \rightarrow \text{if expr then matched\_stmt else matched\_stmt}$   
 $\quad \mid \text{other}$   
 $\text{unmatched\_stmt} \rightarrow \text{if expr then Stmt}$   
 $\quad \mid \text{if expr then matched\_stmt else unmatched\_stmt}$

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

### 3. Loại bỏ đệ qui trái

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

Đệ qui trái có hai loại :

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

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

Xét văn phạm như sau:

$$S \rightarrow Aa \mid b$$

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

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

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

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

Sẽ thay thế bởi :

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

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

#### □ Giải thuật 4.1: Loại bỏ đệ qui trái

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

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

**Phương pháp:**

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

**Begin**

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

**begin**

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

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

**end;**

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

**End;**

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

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

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

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

$S \rightarrow Aa \mid b$

$A \rightarrow bdA'$

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

#### 4. Tạo ra yếu tố trái

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

Xét văn phạm cho câu lệnh if:

$\text{stmt} \rightarrow \text{if expr then stmt else stmt}$   
 $\quad \mid \text{if expr then stmt}$



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

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

#### □ Giải thuật 4.2 : Tạo yếu tố trái cho văn phạm

**Input:** Văn phạm G

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

**Phương pháp:**

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

$$\begin{cases} A \rightarrow \alpha A' \mid \gamma \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{cases}$$

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

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

$$S \rightarrow iEtS \mid iEtSeS \mid \alpha$$

$$E \rightarrow b$$

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

$$S \rightarrow iEtSS' \mid \alpha$$

$$S' \rightarrow eS \mid \varepsilon$$

$$E \rightarrow b$$

### III. PHÂN TÍCH CÚ PHÁP TỪ TRÊN XUỐNG

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

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

Phân tích cú pháp từ trên xuống có thể được xem như một nỗ lực tìm kiếm một dẫn xuất trái nhất cho chuỗi nhập. Nó cũng có thể xem như một nỗ lực xây dựng cây phân tích cú pháp bắt đầu từ nút gốc và phát sinh dần xuống lá. Một dạng tổng quát của kỹ thuật phân tích từ trên xuống, gọi là **phân tích cú pháp đệ quy lùi**, có thể quay lui để

quét lại chuỗi nhập. Tuy nhiên, dạng này thường rất ít gặp. Lý do là với các kết cấu ngôn ngữ lập trình, chúng ta hiếm khi dùng đến nó.

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

Trong nhiều trường hợp, bằng cách viết văn phạm một cách cẩn thận, loại bỏ đệ qui trái ra khỏi văn phạm rồi tạo ra yếu tố trái, chúng ta có thể thu được một văn phạm mà một bộ phân tích cú pháp đệ quy lùi phân tích được, nhưng không cần quay lui, gọi là phân tích cú pháp dự đoán.

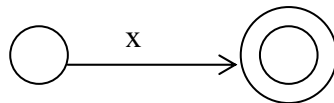
### Xây dựng sơ đồ dịch cho bộ phân tích dự đoán:

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

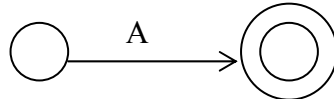
1. Tạo một trạng thái khởi đầu và một trạng thái kết thúc.
2. Với mỗi luật sinh  $A \rightarrow X_1 X_2 \dots X_n$ , tạo một đường đi từ trạng thái khởi đầu đến trạng thái kết thúc bằng các cạnh có nhãn  $X_1 X_2 \dots X_n$

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

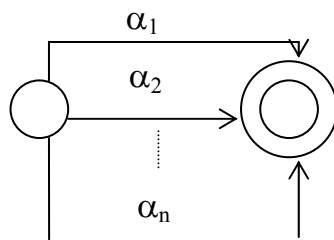
1. Mỗi ký hiệu chưa kết thúc tương ứng với một sơ đồ dịch trong đó nhãn cho các cạnh là token hoặc ký hiệu chưa kết thúc.
2. Mỗi token tương ứng với việc đoán nhận token đó và đọc token kế tiếp



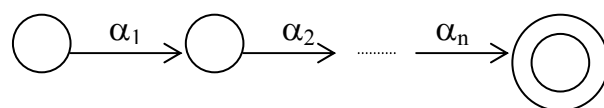
3. Mỗi ký hiệu chưa kết thúc tương ứng với lời gọi thủ tục cho ký hiệu đó.



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



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



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

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

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

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

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

$$E \rightarrow TE'$$

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

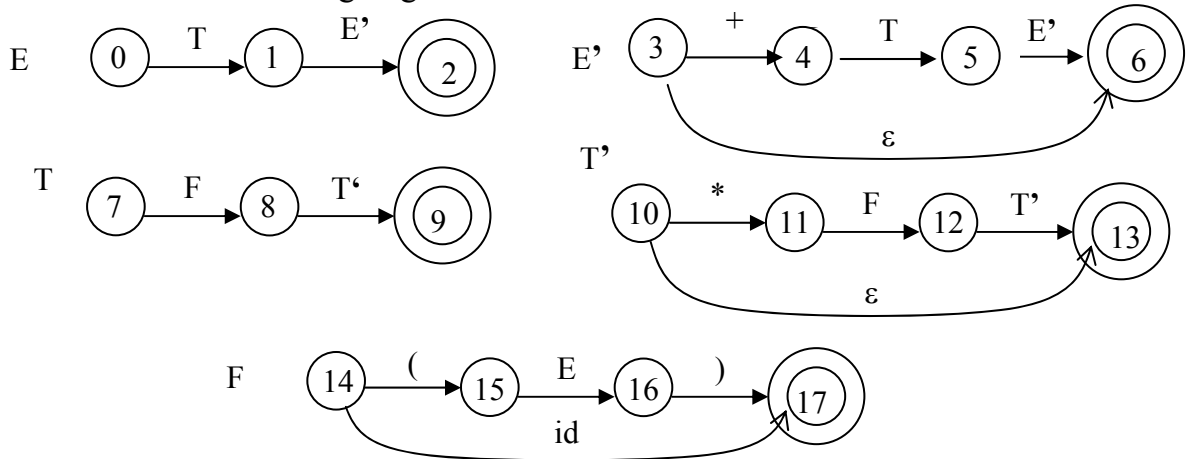
$$T \rightarrow FT'$$

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

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

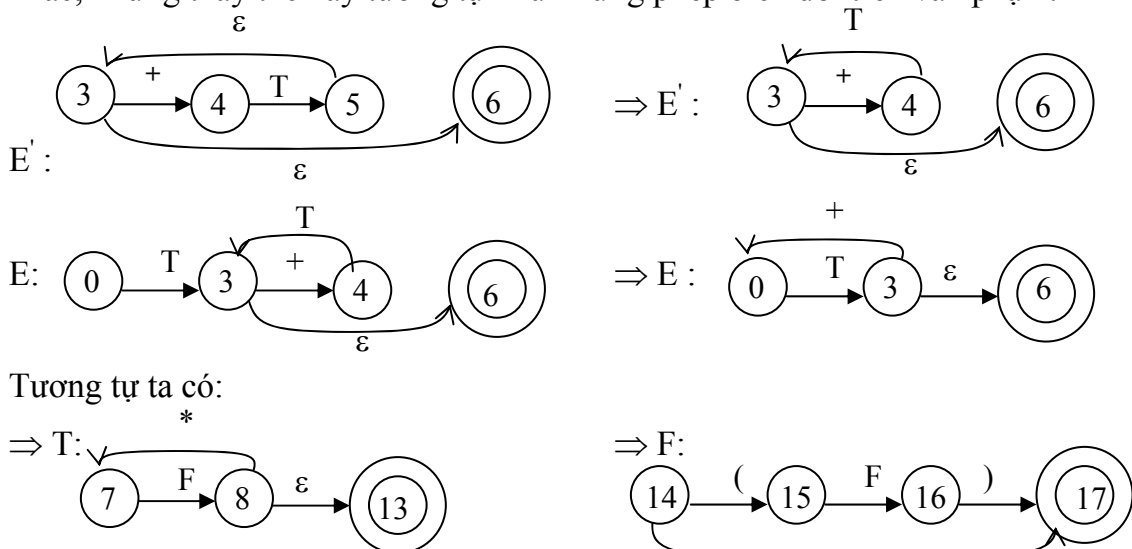
Một chương trình phân tích cú pháp dự đoán được thiết kế dựa trên sơ đồ dịch cho các ký hiệu chưa kết thúc trong văn phạm. Nó sẽ cố gắng so sánh các ký hiệu kết thúc với chuỗi nguyên liệu và đưa ra lời gọi đệ qui mỗi khi nó phải đi theo một cạnh có nhãn là ký hiệu chưa kết thúc.

Các sơ đồ dịch tương ứng :



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

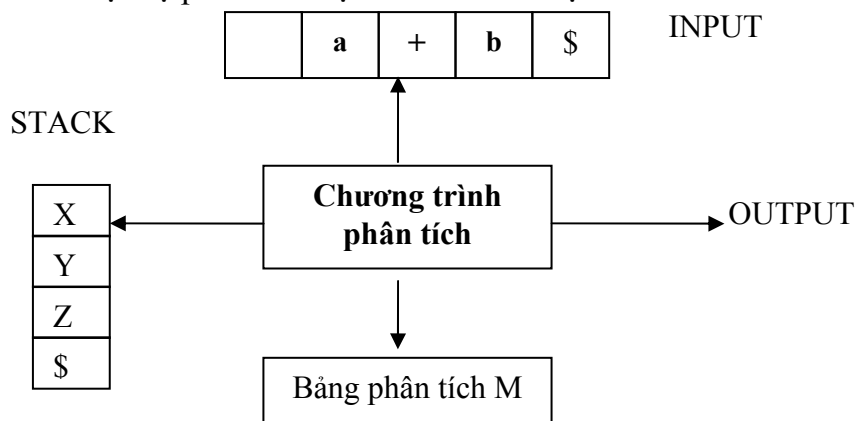
Các sơ đồ dịch có thể được đơn giản hóa bằng cách thay sơ đồ này vào sơ đồ khác, những thay thế này tương tự như những phép biến đổi trên văn phạm.



**Hình 4.6 - Rút gọn sơ đồ dịch**

**Phân tích dự đoán không đệ qui**

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



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

- **INPUT** là bộ đệm chứa chuỗi cần phân tích, kết thúc bởi ký hiệu \$.
- **STACK** chứa một chuỗi các ký hiệu văn phạm với ký hiệu \$ nằm ở đáy Stack.
- **Bảng phân tích M** là một mảng hai chiều dạng  $M[A,a]$ , trong đó A là ký hiệu chưa kết thúc, a là ký hiệu kết thúc hoặc \$.

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

1. Nếu  $X = a = \$$  thì chương trình phân tích cú pháp kết thúc thành công.
2. Nếu  $X = a \neq \$$ , Pop X ra khỏi Stack và đọc ký hiệu nhập tiếp theo.
3. Nếu X là ký hiệu chưa kết thúc thì chương trình truy xuất đến phần tử  $M[X,a]$  trong bảng phân tích M:
  - Nếu  $M[X,a]$  là một luật sinh có dạng  $X \rightarrow UVW$  thì Pop X ra khỏi đỉnh Stack và Push W, V, U vào Stack (với U trên đỉnh Stack), đồng thời bộ xuất sinh ra luật sinh  $X \rightarrow UVW$ .
  - Nếu  $M[X,a] = \text{error}$ , gọi chương trình phục hồi lỗi.

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

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

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

**Phương pháp:**

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

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

**Repeat**

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

**If** X là ký hiệu kết thúc hoặc \$ **then**  
    **If** X = a **then** lấy X ra khỏi Stack và dịch chuyển ip  
    **else** error ( )  
**Else** // X là ký hiệu chưa kết thúc  
    **If**  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  **then**  
        **begin**  
            Lấy X ra khỏi Stack;  
            Đẩy  $Y_k, Y_{k-1}, \dots, Y_1$  vào Stack;  
            Xuất ra luật sinh  $X \rightarrow Y_1 Y_2 \dots Y_k$ ;  
        **end**  
    **else** error ( )     /\* Stack rỗng \*/  
**Until**     X = \$

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

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

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

Ký hiệu chưa kết thúc	Ký hiệu nhập					
	id	+	*	(	)	\$
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<b>E'</b>		$E \rightarrow +TE'$			$E \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
<b>T</b>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<b>T'</b>		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
<b>F</b>	$F \rightarrow id$			$F \rightarrow (E)$		

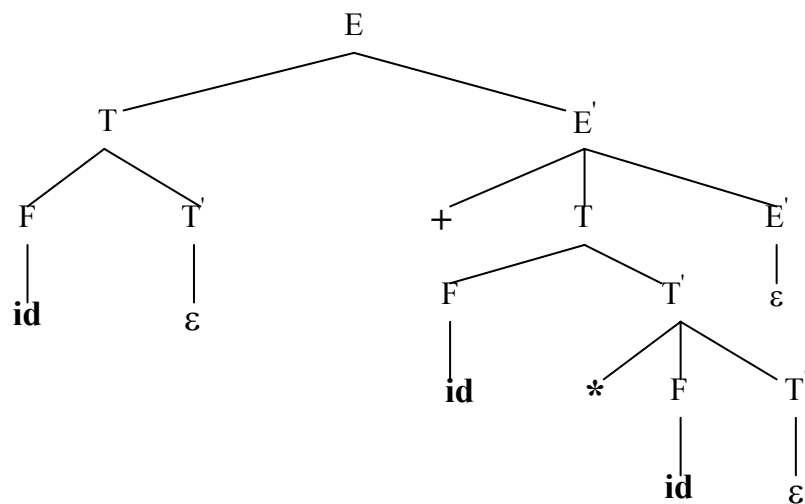
**Hình 4.8** - Bảng phân tích cú pháp M cho văn phạm

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

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

\$ E' T' F	id + id * id \$	$T \rightarrow F T'$
\$ E' T' id	id + id * id \$	$F \rightarrow id$
\$ E' T'	+ id * id \$	
\$ E'	+ id * id \$	$T' \rightarrow \varepsilon$
\$ E' T +	+ id * id \$	$E' \rightarrow + T E'$
\$ E' T	id * id \$	
\$ E' T' F	id * id \$	$T \rightarrow F T'$
\$ E' T' id	id * id \$	$F \rightarrow id$
\$ E' T'	* id \$	
\$ E' T' F *	* id \$	$T' \rightarrow * F T'$
\$ E' T' F	id \$	$F \rightarrow id$
\$ E' T' id	id \$	
\$ E' T'	\$	$T' \rightarrow \varepsilon$
\$ E'	\$	$E' \rightarrow \varepsilon$
\$	\$	

Cây phân tích cú pháp được hình thành từ output :



**Nhận xét:**

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

### 3. Hàm FIRST và FOLLOW

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

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

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

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

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

**Ví dụ 4.7:** Với văn phạm sau:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \varepsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \varepsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Theo định nghĩa tập FIRST, ta có :

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

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

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

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

$$\text{Mặt khác do } E' \rightarrow + T E' \text{ mà } \text{FIRST}(+) = \{+\} \Rightarrow \text{FIRST}(E') = \{+, \varepsilon\}$$

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

Vậy ta có :

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

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

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

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

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

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

2. Nếu có một luật sinh  $A \rightarrow \alpha B \beta$  thì thêm mọi phần tử khác  $\varepsilon$  của  $FIRST(\beta)$  vào trong  $FOLLOW(B)$ .
3. Nếu có luật sinh  $A \rightarrow \alpha B$  hoặc  $A \rightarrow \alpha B \beta$  mà  $\varepsilon \in FIRST(\beta)$  thì thêm tất cả các phần tử trong  $FOLLOW(A)$  vào  $FOLLOW(B)$ .

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

Áp dụng luật 2 cho luật sinh  $F \rightarrow (E) \Rightarrow ) \in FOLLOW(E) \Rightarrow FOLLOW(E) = \{ \$, ) \}$

Áp dụng luật 3 cho  $E \rightarrow TE' \Rightarrow ), \$ \in FOLLOW(E') \Rightarrow FOLLOW(E') = \{ \$, ) \}$

Áp dụng luật 2 cho  $E \rightarrow TE' \Rightarrow + \in FOLLOW(T)$ .

Áp dụng luật 3 cho  $E' \rightarrow +TE', E' \rightarrow \varepsilon$

$\Rightarrow FOLLOW(E') \subset FOLLOW(T) \Rightarrow FOLLOW(T) = \{ +, ), \$ \}$ .

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

Áp dụng luật 2 cho  $T \rightarrow FT' \Rightarrow * \in FOLLOW(F)$

Áp dụng luật 3 cho  $T' \rightarrow *FT', T' \rightarrow \varepsilon$

$\Rightarrow FOLLOW(T') \subset FOLLOW(F) \Rightarrow FOLLOW(F) = \{ *, +, ), \$ \}$ .

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

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

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

#### 4. Xây dựng bảng phân tích M

##### □ Giải thuật 4.4 : Xây dựng bảng phân tích cú pháp dự đoán

**Input:** Văn phạm G.

**Output:** Bảng phân tích cú pháp M.

**Phương pháp:**

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

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

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

$\Rightarrow M[E, id]$  và  $M[E, ( ]$  chứa luật sinh  $E \rightarrow TE'$

Luật sinh  $E' \rightarrow +TE'$  : Tính  $FIRST(+TE') = FIRST(+ ) = \{ + \}$

$\Rightarrow M[E', +]$  chứa  $E' \rightarrow +TE'$



Luật sinh  $E' \rightarrow \varepsilon$  : Vì  $\varepsilon \in \text{FIRST}(E')$  và  $\text{FOLLOW}(E') = \{ ), \$ \}$

$\Rightarrow E \rightarrow \varepsilon$  nằm trong  $M[E', )]$  và  $M[E', \$]$

Luật sinh  $T' \rightarrow * FT'$  :  $\text{FIRST}(* FT') = \{ * \}$

$\Rightarrow T' \rightarrow * FT'$  nằm trong  $M[T', *]$

Luật sinh  $T' \rightarrow \varepsilon$  : Vì  $\varepsilon \in \text{FIRST}(T')$  và  $\text{FOLLOW}(T') = \{ +, ), \$ \}$

$\Rightarrow T' \rightarrow \varepsilon$  nằm trong  $M[T', +]$ ,  $M[T', )]$  và  $M[T', \$]$

Luật sinh  $F \rightarrow (E)$  ;  $\text{FIRST}((E)) = \{ ( \}$

$\Rightarrow F \rightarrow (E)$  nằm trong  $M[F, (]$

Luật sinh  $F \rightarrow \text{id}$  ;  $\text{FIRST}(\text{id}) = \{ \text{id} \}$

$\Rightarrow F \rightarrow \text{id}$  nằm trong  $M[F, \text{id}]$

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

## 5. Văn phạm LL(1)

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

**Ví dụ 4.10:** Xét văn phạm sau:

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

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

Ký hiệu chưa kết thúc	Ký hiệu kết thúc					
	a	b	e	i	T	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S \rightarrow \varepsilon$ $S' \rightarrow eS$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

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

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

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

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

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

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

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

1. Không có một ký hiệu kết thúc a nào mà cả  $\alpha$  và  $\beta$  đều dẫn xuất ra các chuỗi bắt đầu bằng a.
2. Tối đa chỉ có  $\alpha$  hoặc chỉ có  $\beta$  có thể dẫn xuất ra chuỗi rỗng.
3. Nếu  $\beta \Rightarrow^* \epsilon$  thì  $\alpha$  không dẫn xuất được chuỗi nào bắt đầu bằng một ký hiệu kết thúc thuộc tập FOLLOW(A).

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

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

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

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

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

1. Ký hiệu kết thúc trên đỉnh Stack không phù hợp với token kế tiếp trong dòng nhập. Hoặc :
2. Trên đỉnh Stack là ký hiệu chưa kết thúc A, token trong dòng nhập là a nhưng  $M[A, a]$  rỗng.

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

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

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

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

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

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

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

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

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

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

Ký hiệu chưa kết thúc	Ký hiệu kết thúc					
	id	+	*	(	)	\$
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$	<b>synch</b>	<b>synch</b>
<b>E'</b>		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
<b>T</b>	$T \rightarrow FT'$	<b>synch</b>		$T \rightarrow FT'$	<b>synch</b>	<b>synch</b>
<b>T'</b>		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
<b>F</b>	$F \rightarrow id$	<b>synch</b>	<b>synch</b>	$F \rightarrow (E)$	<b>synch</b>	<b>synch</b>

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

Bảng này được sử dụng như sau:

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

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

STACK	INPUT	OUTPUT
\$ E	+ id * + id \$	<b>error, nhảy qua +</b>
\$ E	id * + id \$	$E \rightarrow T E'$
\$ E' T	id * + id \$	$T \rightarrow F T'$

\$ E' T' F	id * + id \$	$F \rightarrow id$
\$ E' T' id	id * + id \$	
\$ E' T'	* + id \$	$T' \rightarrow * F T'$
\$ E' T' F *	* + id \$	
\$ E' T' F	+ id \$	<b>error</b> , $M[F, +] = \text{synch pop } F$
\$ E' T'	+ id \$	$T \rightarrow \varepsilon$
\$ E'	+ id \$	$E' \rightarrow + T E'$
\$ E' T +	+ id \$	
\$ E' T	id \$	$T' \rightarrow F T'$
\$ E' T' F	id \$	$F \rightarrow id$
\$ E' T' id	id \$	$T' \rightarrow \varepsilon$
\$ E' T'	\$	$E' \rightarrow \varepsilon$
\$ E'	\$	
\$	\$	

#### IV. PHÂN TÍCH CÚ PHÁP TỪ DƯỚI LÊN

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

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

Phân tích cú pháp Shift - Reduce cố gắng xây dựng một cây phân tích cú pháp cho chuỗi nhập bắt đầu từ nút lá và đi lên hướng về nút gốc. Đây có thể xem là quá trình thu gọn (reduce) một chuỗi  $w$  thành một ký hiệu bắt đầu của văn phạm. Tại mỗi bước thu gọn, một chuỗi con cụ thể đối sánh được với vế phải của một luật sinh nào đó thì chuỗi con này sẽ được thay thế bởi ký hiệu vế trái của luật sinh đó. Và nếu chuỗi con được chọn đúng tại mỗi bước, một dẫn xuất phải đảo ngược sẽ được xây dựng.

**Ví dụ 4.12:** Cho văn phạm:

$$S \rightarrow a A B e$$

$$A \rightarrow A b c \mid b$$

$$B \rightarrow d$$

Câu  $abbcd e$  có thể thu gọn thành  $S$  theo các bước sau:

$$a \underline{b} b c d e$$

$$a \underline{A b c} d e$$

$$a A \underline{d} e$$

$$\frac{a A B e}{S}$$

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

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

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

## 2. Handle

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

**Ví dụ 4.13:** Xét văn phạm sau:

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

Chuỗi dẫn xuất phải :

$$\begin{aligned} E &\Rightarrow_{rm} E + E && \text{(các handle được gạch dưới)} \\ &\Rightarrow_{rm} E + \underline{E * E} \\ &\Rightarrow_{rm} E + E * \underline{id_3} \\ &\Rightarrow_{rm} E + \underline{id_2} * \underline{id_3} \\ &\Rightarrow_{rm} \underline{id_1} + \underline{id_2} * \underline{id_3} \end{aligned}$$

## 3. Cắt tỉa handle (Handle Pruning)

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

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

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

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

**Ví dụ 4.14:** Với văn phạm:

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

Và câu nhập:  $id_1 + id_2 * id_3$ , ta có các bước thu gọn câu nhập thành ký hiệu bắt đầu  $E$  như sau :

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

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

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

##### Cấu tạo:

Dùng 1 Stack để lưu các ký hiệu văn phạm.

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

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

##### Hoạt động:

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

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

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

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

\$ E + E * E	\$	Thu gọn bởi E → id
\$ E + E	\$	Thu gọn bởi E → E * E
\$ E	\$	Thu gọn bởi E → E + E
		Chấp nhận

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

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

### 1. Quan hệ thứ tự ưu tiên

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

Quan hệ	Ý nghĩa
$a <\bullet b$	a có độ ưu tiên thấp hơn b
$a \dot{=} b$	a có độ ưu tiên bằng b
$a \bullet> b$	a có độ ưu tiên cao hơn b

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

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

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

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

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

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

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

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

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

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

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

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

$E - E + E \Rightarrow$  handle là  $E - E$

$E \uparrow E \uparrow E \Rightarrow$  handle là  $E \uparrow E$  (phần cuối)

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

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

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

Sử dụng \$ để đánh dấu cuối chuỗi và \$ <•  $\theta$ ,  $\forall \theta$ .

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

\$ <• id •> + <• id •> \* <• id •> \$

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

1. Duyệt chuỗi từ trái sang phải cho đến khi gặp •> đầu tiên (trong ví dụ của chúng ta là •> sau id đầu tiên).
2. Sau đó, duyệt ngược lại (hướng sang trái), vượt qua các  $\equiv$  (nếu có) cho đến khi gặp a <• (trong ví dụ, chúng ta quét ngược về đến \$).
3. Handle là chuỗi chứa mọi ký hiệu ở bên trái •> đầu tiên và bên phải <• được gặp trong bước (2), chứa luôn cả các ký hiệu chưa kết thúc xen giữa hoặc bao quanh (trong ví dụ, handle là id đầu tiên).

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

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

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

#### □ Giải thuật 4.5: Phân tích cú pháp thứ bậc toán tử

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

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

**Phương pháp:**

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

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

**Repeat forever**

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

**return**



**Else begin**

**If**  $a < \bullet b$  hoặc  $a = b$  **then begin**

    Đẩy  $b$  vào Stack;

    Dịch ip chỉ đến ký hiệu tiếp theo trong bộ đệm;

**end**

**Else if**  $a \bullet > b$  **then**       /\* thu gọn \*/

**Repeat**

        Lấy ký hiệu trên đỉnh Stack ra;

**Until** Ký hiệu kết thúc trên đỉnh Stack có quan hệ  $< \bullet$  với  
        ký hiệu kết thúc vừa lấy ra;

**else error** ( )

**End**

## VI. BỘ PHÂN TÍCH CÚ PHÁP LR

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

**L** (left - to - right): Duyệt chuỗi nhập từ trái sang phải.

**R** (rightmost derivation): Xây dựng chuỗi dẫn xuất phải nhất đảo ngược.

**k** : Số lượng ký hiệu nhập được xét tại mỗi thời điểm dùng để đưa ra quyết định phân tích. Khi không đề cập đến  $k$ , chúng ta hiểu ngầm là  $k = 1$ .

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

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

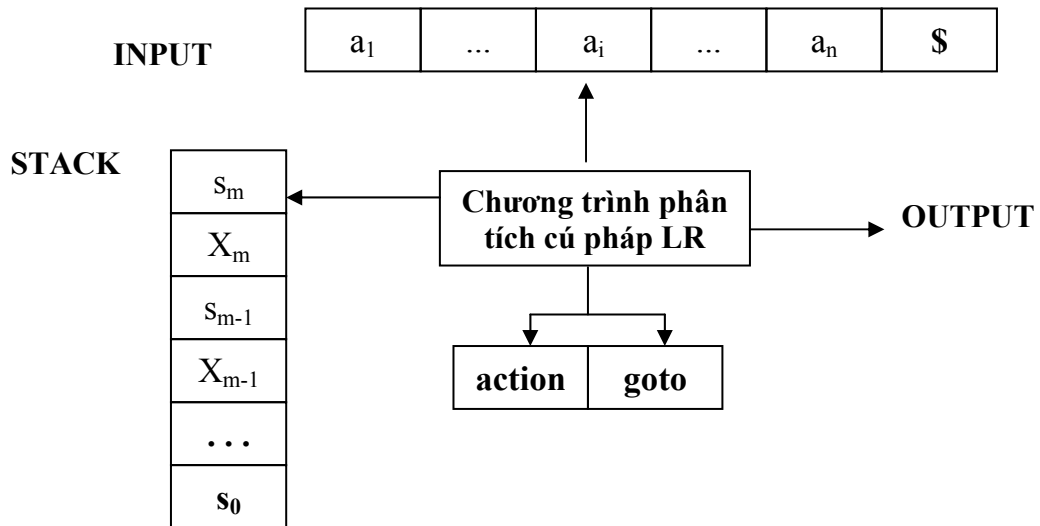
- Phương pháp phân tích cú pháp LR là phương pháp tổng quát của phương pháp chuyên thu gọn không quay lui. Nó có thể được cài đặt có hiệu quả như các phương pháp chuyên thu gọn khác.

- Lớp văn phạm có thể dùng phương pháp LR là một lớp rộng lớn hơn lớp văn phạm có thể sử dụng phương pháp dự đoán.

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

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

## 1. Thuật toán phân tích cú pháp LR



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

- **Stack** lưu một chuỗi  $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$  trong đó  $s_m$  nằm trên đỉnh Stack.  $X_i$  là một ký hiệu văn phạm,  $s_i$  là một trạng thái tóm tắt thông tin chứa trong Stack bên dưới nó.
- **Bảng phân tích** bao gồm 2 phần: hàm action và hàm goto.
  - ✓ **action** $[s_m, a_i]$  có thể có một trong 4 giá trị :
    1. **shift s**: đẩy s, trong đó s là một trạng thái.
    2. **reduce  $A \rightarrow \beta$** : thu gọn bằng luật sinh  $A \rightarrow \beta$ .
    3. **accept**: Chấp nhận
    4. **error**: Báo lỗi
  - ✓ **Goto** lấy 2 tham số là một trạng thái và một ký hiệu văn phạm, nó sinh ra một trạng thái.

**Cấu hình** (configuration) của một bộ phân tích cú pháp LR là một cặp thành phần, trong đó, thành phần đầu là nội dung của Stack, phần sau là chuỗi nhập chưa phân tích:  $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$

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

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

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$$

Phép đẩy làm cho s nằm trên đỉnh Stack,  $a_{i+1}$  trở thành ký hiệu hiện hành.

2. Nếu **action**  $[s_m, a_i] = \text{Reduce}(A \rightarrow \beta)$  thì thực hiện phép thu gọn để được cấu hình:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_{m-i} s_{m-i} A s, a_i a_{i+1} \dots a_n \$)$$

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

3. Nếu  $\text{action}[s_m, a_i] = \text{accept}$ : quá trình phân tích kết thúc.

4. Nếu  $\text{action}[s_m, a_i] = \text{error}$ : gọi thủ tục phục hồi lỗi.

#### □ Giải thuật 4.6 : Phân tích cú pháp LR

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

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

#### Phương pháp:

Khởi tạo  $s_0$  là trạng thái khởi tạo nằm trong Stack và  $w\$$  nằm trong bộ đệm nhập.

Đặt  $ip$  vào ký hiệu đầu tiên của  $w\$$ ;

#### Repeat forever begin

Gọi  $s$  là trạng thái trên đỉnh Stack và  $a$  là ký hiệu được trỏ bởi  $ip$ ;

**If**  $\text{action}[s, a] = \text{Shift } s'$  **then begin**

    Đẩy  $a$  và sau đó là  $s'$  vào Stack;

    Chuyển  $ip$  tới ký hiệu kế tiếp;

**end**

**else if**  $\text{action}[s, a] = \text{Reduce } (A \rightarrow \beta)$  **then begin**

    Lấy  $2 * |\beta|$  ký hiệu ra khỏi Stack;

    Gọi  $s'$  là trạng thái trên đỉnh Stack;

    Đẩy  $A$ , sau đó đẩy  $\text{goto}[s', A]$  vào Stack;

    Xuất ra luật sinh  $A \rightarrow \beta$ ;

**end**

**else if**  $\text{action}[s, a] = \text{accept}$  **then**

**return**

**else error** ( )

**end**

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

(1)  $E \rightarrow E + T$

(2)  $E \rightarrow T$

(3)  $T \rightarrow T * F$

(4)  $T \rightarrow F$

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

(5)  $F \rightarrow (E)$

(6)  $F \rightarrow id$

**Ý nghĩa:**

**si** : chuyển trạng thái i

**ri** : thu gọn bởi luật sinh i

**acc**: accept (chấp nhận)

**error** : khoảng trống

<b>2</b>		r <sub>2</sub>	s <sub>7</sub>		r <sub>2</sub>	r <sub>2</sub>			
<b>3</b>		r <sub>4</sub>	r <sub>4</sub>		r <sub>4</sub>	r <sub>4</sub>			
<b>4</b>	s <sub>5</sub>			s <sub>4</sub>			8	2	3
<b>5</b>		r <sub>6</sub>	r <sub>6</sub>		r <sub>6</sub>	r <sub>6</sub>			
<b>6</b>	s <sub>5</sub>			s <sub>4</sub>				9	3
<b>7</b>	s <sub>5</sub>			s <sub>4</sub>					1 0
<b>8</b>		s <sub>6</sub>			s <sub>11</sub>				
<b>9</b>		r <sub>1</sub>	s <sub>7</sub>		r <sub>1</sub>	r <sub>1</sub>			
<b>10</b>		r <sub>3</sub>	r <sub>3</sub>		r <sub>3</sub>	r <sub>3</sub>			
<b>11</b>		r <sub>5</sub>	r <sub>5</sub>		r <sub>5</sub>	r <sub>5</sub>			

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

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

STACK	INPUT	ACTION
(1) 0	id * id + id \$	Shift
(2) 0 id 5	* id + id \$	Reduce by $F \rightarrow id$
(3) 0 F 3	* id + id \$	Reduce by $T \rightarrow F$
(4) 0 T 2	* id + id \$	Shift
(5) 0 T 2 * 7	id + id \$	Shift
(6) 0 T 2 * 7 id 5	+ id \$	Reduce by $F \rightarrow id$
(7) 0 T 2 * 7 F 10	+ id \$	Reduce by $T \rightarrow T * F$
(8) 0 T 2	+ id \$	Reduce by $E \rightarrow T$
(9) 0 E 1	+ id \$	Shift
(10) 0 E 1 + 6	id \$	Shift
(11) 0 E 1 + 6 id 5	\$	Reduce by $F \rightarrow id$
(12) 0 E 1 + 6 F 3	\$	Reduce by $T \rightarrow F$
(13) 0 E 1 + 6 T 9	\$	Reduce by $E \rightarrow E + T$
(14) 0 E 1	\$	<b>Thành công</b>

## 2. Văn phạm LR

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

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

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

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

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

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

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

$$A \rightarrow \bullet XYZ$$

$$A \rightarrow X \bullet YZ$$

$$A \rightarrow XY \bullet Z$$

$$A \rightarrow XYZ \bullet$$

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

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

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

#### c. Phép toán bao đóng (Closure)

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

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

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

$$E' \rightarrow E$$

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

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

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

Nếu  $I$  là tập hợp chỉ gồm văn phạm  $\{ E' \rightarrow \bullet E \}$  thì  $\text{closure}(I)$  bao gồm:

$$E' \rightarrow \bullet E \quad (\text{Luật 1})$$

$$E \rightarrow \bullet E + T \quad (\text{Luật 2})$$

$$E \rightarrow \bullet T \quad (\text{Luật 2})$$

$$T \rightarrow \bullet T * F \quad (\text{Luật 2})$$

$$T \rightarrow \bullet F \quad (\text{Luật 2})$$

$$F \rightarrow \bullet (E) \quad (\text{Luật 2})$$

$$F \rightarrow \bullet id \quad (\text{Luật 2})$$

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

#### d. Phép toán Goto

$\text{Goto}(I, X)$ , trong đó  $I$  là một tập các mục và  $X$  là một ký hiệu văn phạm là bao đóng của tập hợp các mục  $A \rightarrow \alpha X \bullet \beta$  sao cho  $A \rightarrow \alpha \bullet X \beta \in I$ .

Cách tính  $\text{goto}(I, X)$ :

1. Tạo một tập  $I' = \emptyset$ .
2. Nếu  $A \rightarrow \alpha \bullet X \beta \in I$  thì đưa  $A \rightarrow \alpha X \bullet \beta$  vào  $I'$ , tiếp tục quá trình này cho đến khi xét hết tập  $I$ .
3.  $\text{Goto}(I, X) = \text{closure}(I')$

**Ví dụ 4.21:** Giả sử  $I = \{ E' \rightarrow E \bullet, E \rightarrow E \bullet + T \}$ .

Tính  $\text{goto}(I, +)$  ?

Ta có  $I' = \{ E \rightarrow E + \bullet T \}$

(  $\text{goto}(I, +) = \text{closure}(I')$  bao gồm các mục :

$$E \rightarrow E + \bullet T \quad (\text{Luật 1})$$

$$T \rightarrow \bullet T * F \quad (\text{Luật 2})$$

$$T \rightarrow \bullet F \quad (\text{Luật 2})$$

$$F \rightarrow \bullet (E) \quad (\text{Luật 2})$$

$$F \rightarrow \bullet id \quad (\text{Luật 2})$$

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

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

**Procedure** Item ( $G'$ )

**begin**

$C := \text{closure}(\{ S' \rightarrow \bullet S \});$

**Repeat**

**For** Với mỗi tập các mục I trong C và mỗi ký hiệu văn phạm X

sao cho  $\text{goto}(I, X) \neq \emptyset$  và  $\text{goto}(I, X) \notin C$  do

Thêm  $\text{goto}(I, X)$  vào C;

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

**end;**

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

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

$\text{Closure}(\{E' \rightarrow E\})$ :

	<b>I<sub>0</sub>:</b>	$E' \rightarrow \bullet E$			
		$E \rightarrow \bullet E + T$	$\text{Goto}(I_0, \text{id})$	<b>I<sub>5</sub>:</b>	$F \rightarrow \text{id} \bullet$
		$E \rightarrow \bullet T$			
		$T \rightarrow \bullet T * F$	$\text{Goto}(I_1, +)$	<b>I<sub>6</sub>:</b>	$E \rightarrow E + \bullet T$
		$T \rightarrow \bullet F$			$T \rightarrow \bullet T * F$
		$F \rightarrow \bullet (E)$			$T \rightarrow \bullet F$
		$F \rightarrow \bullet \text{id}$			$F \rightarrow \bullet (E)$
					$F \rightarrow \bullet \text{id}$
$\text{Goto}(I_0, E)$	<b>I<sub>1</sub>:</b>	$E' \rightarrow E \bullet$			
		$E \rightarrow E \bullet + T$	$\text{Goto}(I_2, *)$	<b>I<sub>7</sub>:</b>	$T \rightarrow T * \bullet F$
					$F \rightarrow \bullet (E)$
$\text{Goto}(I_0, T)$	<b>I<sub>2</sub>:</b>	$E \rightarrow T \bullet$			$F \rightarrow \bullet \text{id}$
		$T \rightarrow T \bullet * F$			
			$\text{Goto}(I_4, E)$	<b>I<sub>8</sub>:</b>	$T \rightarrow (E \bullet)$
$\text{Goto}(I_0, F)$	<b>I<sub>3</sub>:</b>	$T \rightarrow F \bullet$			$E \rightarrow E \bullet + T$
$\text{Goto}(I_0, ($	<b>I<sub>4</sub>:</b>	$F \rightarrow (\bullet E)$	$\text{Goto}(I_6, T)$	<b>I<sub>9</sub>:</b>	$E \rightarrow E + T \bullet$
		$E \rightarrow \bullet E + T$			$T \rightarrow T \bullet * F$
		$E \rightarrow \bullet T$			
		$T \rightarrow \bullet T * F$	$\text{Goto}(I_7, F)$	<b>I<sub>10</sub>:</b>	$T \rightarrow T * F \bullet$
		$T \rightarrow \bullet F$			
		$F \rightarrow \bullet (E)$	$\text{Goto}(I_8, )$	<b>I<sub>11</sub>:</b>	$F \rightarrow (E) \bullet$
		$F \rightarrow \bullet \text{id}$			

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

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

**Input:** Một văn phạm tăng cường  $G'$

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

**Phương pháp:**

1. Xây dựng  $C = \{ I_0, I_1, \dots, I_n \}$ , họ tập hợp các mục LR(0) của  $G'$ .
2. Trạng thái  $i$  được xây dựng từ  $I_i$ . Các action tương ứng trạng thái  $i$  được xác định như sau:

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

2.2. Nếu  $A \rightarrow \alpha \bullet \in I_i$  thì  $\text{action}[i, a] = \text{"reduce (A} \rightarrow \alpha\text{)"}$ ,  $\forall a \in \text{FOLLOW}(A)$ . Ở đây  $A$  không phải là  $S'$

2.3. Nếu  $S' \rightarrow S \bullet \in I_i$  thì  $\text{action}[i, \$] = \text{"accept"}$ .

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

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

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

5. Trạng thái khởi đầu của bộ phân tích cú pháp được xây dựng từ tập các mục chứa  $S' \rightarrow \bullet S$

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

$E' \rightarrow E$	(0) $E' \rightarrow E$	(4) $T \rightarrow F$
$E \rightarrow E + T \mid T$	(1) $E \rightarrow E + T$	(5) $F \rightarrow (E)$
$T \rightarrow T * F \mid F$	(2) $E \rightarrow T$	(6) $F \rightarrow \text{id}$
$F \rightarrow (E) \mid \text{id}$	(3) $T \rightarrow T * F$	

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

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

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

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

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

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

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

Kế đến xét  $I_2$ :  $E \rightarrow T \bullet$



$$T \rightarrow T \bullet * F$$

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

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

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

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

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

$$S \rightarrow L = R$$

$$S \rightarrow R$$

$$L \rightarrow * R$$

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

$$R \rightarrow L$$

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

Họ tập hợp các mục C bao gồm:

$$I_0 : S' \rightarrow \bullet S$$

$$S \rightarrow \bullet L = R$$

$$S \rightarrow \bullet R$$

$$L \rightarrow \bullet * R$$

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

$$R \rightarrow \bullet L$$

$$I_1 : S' \rightarrow S \bullet$$

$$I_2 : S \rightarrow L \bullet = R$$

$$R \rightarrow L \bullet$$

$$I_3 : S \rightarrow R \bullet$$

$$I_4 : L \rightarrow * \bullet R$$

$$R \rightarrow \bullet L$$

$$L \rightarrow \bullet * R$$

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

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

$$I_6 : S \rightarrow L = \bullet R$$

$$R \rightarrow \bullet L$$

$$L \rightarrow \bullet * R$$

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

$$I_7 : L \rightarrow * R \bullet$$

$$I_8 : R \rightarrow L \bullet$$

$$I_9 : S \rightarrow L = R \bullet$$

Khi xây dựng bảng phân tích SLR cho văn phạm, khi xét tập mục  $I_2$  ta thấy mục đầu tiên trong tập này làm cho  $\text{action}[2, =] = \text{"shift 6"}$ . Bởi vì  $= \in \text{FOLLOW}(R)$ , nên mục thứ hai sẽ đặt  $\text{action}[2, =] = \text{"reduce (R} \rightarrow \text{L)"}$   $\Rightarrow$  Có sự ðụng ðộ tại  $\text{action}[2, =]$ . Vậy văn phạm trên không là văn phạm SLR(1).

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

**a. Mục LR(1)** của văn phạm  $G$  là một cặp dạng  $[A \rightarrow \alpha \bullet \beta, a]$ , trong đó  $A \rightarrow \alpha \beta$  là luật sinh,  $a$  là một ký hiệu kết thúc hoặc \$.

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

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

**Input :** Văn phạm tăng cường  $G'$

**Output:** Họ tập hợp các mục LR(1).

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

**Function Closure (I);**

**begin**

*Repeat*

*For* Mỗi mục  $[A \rightarrow \alpha \bullet B \beta, a]$  trong  $I$ , mỗi luật sinh  $B \rightarrow \gamma$  trong  $G'$  và mỗi ký hiệu kết thúc  $b \in \text{FIRST}(\beta a)$  sao cho  $[B \rightarrow \bullet \gamma, b] \notin I$  **do**

Thêm  $[B \rightarrow \bullet \gamma, b]$  vào  $I$ ;

*Until* Không còn mục nào có thể thêm cho  $I$  được nữa;

**return**  $I$ ;

**end;**

**Function goto (I, X);**

**begin**

Gọi  $J$  là tập hợp các mục  $[A \rightarrow \alpha X \bullet \beta, a]$  sao cho  $[A \rightarrow \alpha \bullet X \beta, a] \in I$ ;

**return**  $\text{Closure}(J)$ ;

**end;**

**Procedure Items ( $G'$ );**

**begin**

$C := \text{Closure}(\{[S' \rightarrow \bullet S, \$]\})$

*Repeat*

*For* Mỗi tập các mục  $I$  trong  $C$  và mỗi ký hiệu văn phạm  $X$  sao cho  $\text{goto}(I, X) \neq \emptyset$  và  $\text{goto}(I, X) \notin C$  **do**

Thêm goto(I, X) vào C;

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

**end;**

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

$S' \rightarrow S$

(1)  $S \rightarrow L = R$

(2)  $S \rightarrow R$

(3)  $L \rightarrow * R$

(4)  $L \rightarrow \text{id}$

(5)  $R \rightarrow L$

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

$I_0 :$   $S' \rightarrow \bullet S, \$$

Closure ( $S' \rightarrow \bullet S, \$$ )  $S \rightarrow \bullet L = R, \$$

$S \rightarrow \bullet R, \$$

$L \rightarrow \bullet * R, = | \$$

$L \rightarrow \bullet \text{id}, = | \$$

$R \rightarrow \bullet L, \$$

Goto ( $I_4, R$ )  $I_7 : L \rightarrow * R \bullet, = | \$$

Goto ( $I_4, L$ )  $I_8 : R \rightarrow L \bullet, = | \$$

Goto ( $I_6, R$ )  $I_9 : S \rightarrow L = R \bullet, \$$

Goto ( $I_6, L$ )  $I_{10} : R \rightarrow L \bullet, \$$

Goto ( $I_0, S$ )  $I_1 : S' \rightarrow S \bullet, \$$

Goto ( $I_0, L$ )  $I_2 : S \rightarrow L \bullet = R, \$$

$R \rightarrow L \bullet, \$$

Goto ( $I_6, *$ )  $I_{11} : L \rightarrow * \bullet R, \$$

$R \rightarrow \bullet L, \$$

$L \rightarrow \bullet * R, \$$

$R \rightarrow \bullet \text{id}, \$$

Goto ( $I_0, R$ )  $I_3 : S \rightarrow R \bullet, \$$

Goto ( $I_0, *$ )  $I_4 : L \rightarrow * \bullet R, = | \$$

$R \rightarrow \bullet L, = | \$$

$L \rightarrow \bullet * R, = | \$$

$R \rightarrow \bullet \text{id}, = | \$$

Goto ( $I_6, \text{id}$ )  $I_{12} : L \rightarrow \text{id} \bullet, \$$

Goto ( $I_{11}, R$ )  $I_{13} : R \rightarrow * R \bullet, \$$

Goto ( $I_0, \text{id}$ )  $I_5 : L \rightarrow \text{id} \bullet, = | \$$

Goto ( $I_{11}, L$ )  $\equiv I_{10}$

$\text{Goto}(I_2, =) \quad I_6 : S \rightarrow L = \bullet R, \$$                        $\text{Goto}(I_{11}, *) \equiv I_{11}$   
 $R \rightarrow \bullet L, \$$   
 $L \rightarrow \bullet * R, \$$                        $\text{Goto}(I_{11}, \text{id}) \equiv I_{12}$   
 $L \rightarrow \bullet \text{id}, \$$

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

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

**Input:** Văn phạm tăng cường  $G'$

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

**Phương pháp:**

1. Xây dựng  $C = \{ I_0, I_1, \dots, I_n \}$  là họ tập hợp mục LR(1)
2. Trạng thái thứ  $i$  được xây dựng từ  $I_i$ . Các action tương ứng trạng thái  $i$  được xác định như sau:
  - 2.1. Nếu  $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$  và  $\text{goto}(I_i, a) = I_j$  thì  $\text{action}[i, a] = \text{"shift } j\text{"}$ . Ở đây  $a$  phải là ký hiệu kết thúc.
  - 2.2. Nếu  $[A \rightarrow \alpha \bullet, a] \in I_i$ ,  $A \in S'$  thì  $\text{action}[i, a] = \text{"reduce } (A \rightarrow \alpha)\text{"}$
  - 2.3. Nếu  $[S' \rightarrow S \bullet, \$] \in I_i$  thì  $\text{action}[i, \$] = \text{"accept"}$ .

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

3. Nếu  $\text{goto}(I_i, A) = I_j$  thì  $\text{goto}[i, A] = j$
4. Tất cả các ô không xác định được bởi 2 và 3 đều là **"error"**
5. Trạng thái khởi đầu của bộ phân tích cú pháp được xây dựng từ tập các mục chứa  $[S' \rightarrow \bullet S, \$]$

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

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

Trạng thái	Action				Goto		
	=	*	id	\$	S	L	R
0		s <sub>4</sub>	s <sub>5</sub>		1	2	3
1				acc			
2	s <sub>6</sub>			r <sub>5</sub>			
3				r <sub>2</sub>			
4		s <sub>4</sub>	s <sub>5</sub>			8	7
5	r <sub>4</sub>						

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

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

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

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

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

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

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

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

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

$I_4$  và  $I_{11}$

$I_5$  và  $I_{12}$

$I_7$  và  $I_{13}$

$I_8$  và  $I_{10}$

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

#### □ Giải thuật 4.10: Xây dựng bảng phân tích LALR

**Input:** Văn phạm tăng cường  $G'$

**Output:** Bảng phân tích LALR

**Phương pháp:**

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

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

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

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

4. Bảng goto được xây dựng như sau

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

**Ví dụ 4.28:** Với ví dụ trên, ta có họ tập hợp mục  $C'$  như sau

$$C' = \{ I_0, I_1, I_2, I_3, I_{411}, I_{512}, I_6, I_{713}, I_{810}, I_9 \}$$

$I_0 : \quad S' \rightarrow \bullet S, \$$ $\text{closure}(S' \rightarrow \bullet S, \$) : S \rightarrow \bullet L = R, \$$ $S \rightarrow \bullet R, \$$ $L \rightarrow \bullet * R, =$ $L \rightarrow \bullet \text{id}, =$ $R \rightarrow \bullet L, \$$	$I_{512} = \text{Goto}(I_0, \text{id}), \text{Goto}(I_6, \text{id}) :$ $L \rightarrow \text{id} \bullet, =   \$$
$I_1 = \text{Goto}(I_0, S) : S' \rightarrow S \bullet, \$$	$I_6 = \text{Goto}(I_2, =) :$ $S \rightarrow L = \bullet R, \$$ $R \rightarrow \bullet L, \$$ $L \rightarrow \bullet * R, \$$ $L \rightarrow \bullet \text{id}, \$$
$I_2 = \text{Goto}(I_0, L) : S \rightarrow L \bullet = R, \$$ $R \rightarrow L \bullet, \$$	$I_{713} = \text{Goto}(I_{411}, R) :$ $L \rightarrow * R \bullet, =   \$$
$I_3 = \text{Goto}(I_0, R) : S \rightarrow R \bullet$	$I_{810} = \text{Goto}(I_{411}, L), \text{Goto}(I_6, L) :$ $R \rightarrow L \bullet, =   \$$
$I_{411} = \text{Goto}(I_0, *), \text{Goto}(I_6, *) :$ $L \rightarrow * \bullet R, =   \$$ $R \rightarrow \bullet L, =   \$$ $L \rightarrow \bullet * R, =   \$$ $R \rightarrow \bullet \text{id}, =   \$$	$I_9 = \text{Goto}(I_6, R) :$ $S \rightarrow L = R \bullet, \$$

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

State	Action				Goto		
	=	*	id	\$	S	L	R
0		S <sub>411</sub>	S <sub>512</sub>		1	2	3
1			acc				
2	S <sub>6</sub>						
3				r <sub>2</sub>			
411						810	713
512	r <sub>4</sub>			r <sub>4</sub>			
6		S <sub>411</sub>	S <sub>512</sub>			810	9
713	r <sub>3</sub>			r <sub>3</sub>			
810	r <sub>5</sub>			r <sub>5</sub>			
9				r <sub>1</sub>			

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

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

## VII. SỬ DỤNG CÁC VĂN PHẠM MƠ HỒ

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

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

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

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

$$E \rightarrow E + E \mid E * E \mid (E) \mid id \quad (1)$$

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

$$\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \text{id}
\end{aligned}
\tag{2}$$

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

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

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

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

$I_0: E' \rightarrow \bullet E$	$\text{Goto}(I_2, E)$	$I_6: E' \rightarrow (E \bullet)$
$E \rightarrow \bullet E + E$		$E \rightarrow E \bullet + E$
$E \rightarrow \bullet E * E$		$E \rightarrow E \bullet * E$
$E \rightarrow \bullet (E)$	$\text{Goto}(I_2, () \equiv I_2$	
$E \rightarrow \bullet \text{id}$	$\text{Goto}(I_2, \text{id}) \equiv I_3$	
$\text{Goto}(I_0, E) \quad I_1: E' \rightarrow E \bullet$	$\text{Goto}(I_4, E)$	$I_7: E \rightarrow E + E \bullet$
$E \rightarrow E \bullet + E$		$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$		$E \rightarrow E \bullet * E$
	$\text{Goto}(I_4, () \equiv I_2$	
$\text{Goto}(I_0, () \quad I_2: E \rightarrow (\bullet E)$	$\text{Goto}(I_4, \text{id}) \equiv I_3$	
$E \rightarrow \bullet E + E$		
$E \rightarrow \bullet E * E$	$\text{Goto}(I_5, E) \quad I_8: E \rightarrow E * E \bullet$	
$E \rightarrow \bullet (E)$		$E \rightarrow E \bullet + E$
$E \rightarrow \bullet \text{id}$		$E \rightarrow E \bullet * E$
	$\text{Goto}(I_5, () \equiv I_2$	
$\text{Goto}(I_0, \text{id}) \quad I_3: E \rightarrow \text{id} \bullet$	$\text{Goto}(I_5, \text{id}) \equiv I_3$	
$\text{Goto}(I_1, +) \quad I_4: E \rightarrow E + \bullet E$	$\text{Goto}(I_6, ))$	$I_9: E \rightarrow (E) \bullet$
$E \rightarrow \bullet E + E$		
$E \rightarrow \bullet E * E$	$\text{Goto}(I_6, +) \equiv I_4$	
$E \rightarrow \bullet (E)$	$\text{Goto}(I_6, *) \equiv I_5$	



$E \rightarrow \bullet id$                        $Goto(I_7, +) \equiv I_4$   
 $Goto(I_1, *)$   $I_5: E \rightarrow E * \bullet E$                        $Goto(I_7, *) \equiv I_5$   
 $E \rightarrow \bullet E + E$                        $Goto(I_8, +) \equiv I_4$   
 $E \rightarrow \bullet E * E$                        $Goto(I_8, *) \equiv I_5$   
 $E \rightarrow \bullet ( E)$   
 $E \rightarrow \bullet id$

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

Trạng thái	Action						Goto
	id	+	*	(	)	\$	E
0	s <sub>3</sub>			s <sub>2</sub>			1
1		s <sub>4</sub>	s <sub>5</sub>			acc	
2	s <sub>3</sub>						6
3		r <sub>4</sub>	r <sub>4</sub>		r <sub>4</sub>	r <sub>4</sub>	
4	s <sub>3</sub>			s <sub>2</sub>			7
5	s <sub>3</sub>			s <sub>2</sub>			8
6		s <sub>4</sub>	s <sub>5</sub>		s <sub>9</sub>		
7		s <sub>4</sub> / r <sub>1</sub>	s <sub>5</sub> / r <sub>1</sub>		r <sub>1</sub>	r <sub>1</sub>	
8		s <sub>4</sub> / r <sub>2</sub>	s <sub>5</sub> / r <sub>2</sub>		r <sub>2</sub>	r <sub>2</sub>	
9		r <sub>3</sub>	r <sub>3</sub>		r <sub>3</sub>	r <sub>3</sub>	

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

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

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

Stack	Input	Output
0	id + id * id \$	
0 id 3	+ id * id \$	Shift s <sub>3</sub>
0 E 1	+ id * id \$	Reduce by $E \rightarrow id$
0 E 1 + 4	id * id \$	Shift s <sub>4</sub>
0 E 1 + 4 id 3	* id \$	Shift s <sub>3</sub>
0 E 1 + 4 E 7	* id \$	Reduce by $E \rightarrow id$

--	--	--

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

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

Stack	Output
0 E 1 + 4 E 7	+ id \$

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

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

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

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

Xét văn phạm cho lệnh điều kiện:

$\text{Stmt} \rightarrow \text{if expr then stmt else stmt}$   
 $\quad \quad \quad | \text{if expr then stmt}$   
 $\quad \quad \quad | \text{other}$

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

$S' \rightarrow S$   
 $S \rightarrow iS eS \quad (1)$   
 $S \rightarrow iS \quad (2)$   
 $S \rightarrow a \quad (3)$

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

$I_0 : S' \rightarrow \bullet S,$   
 $S \rightarrow \bullet iSeS$   
 $S \rightarrow \bullet iS$   
 $S \rightarrow \bullet a$

$Goto(I_0, S) \quad I_1 : S' \rightarrow S \bullet$

$Goto(I_0, i) \quad I_2 : S \rightarrow i \bullet SeS$   
 $S \rightarrow i \bullet S$   
 $S \rightarrow \bullet iSeS$   
 $S \rightarrow \bullet iS$   
 $S \rightarrow \bullet a$

$Goto(I_2, S) \quad I_4 : S \rightarrow iS \bullet eS$   
 $S \rightarrow iS \bullet$

$Goto(I_2, e) \quad I_5 : S \rightarrow iSe \bullet S$   
 $S \rightarrow \bullet iSeS$   
 $S \rightarrow \bullet iS$   
 $S \rightarrow \bullet a$

$Goto(I_5, S) \quad I_6 : S \rightarrow iSeS \bullet$   
 $Goto(I_2, i) \equiv I_2$   
 $Goto(I_2, a) \equiv I_3$   
 $Goto(I_5, i) \equiv I_2$   
 $Goto(I_5, a) \equiv I_3$

$Goto(I_0, a) \quad I_3 : S \rightarrow a \bullet$

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

Trạng thái	Action				Goto
	i	e	a	\$	S
<b>0</b>	$s_2$		$s_3$		<b>1</b>
<b>1</b>				<b>acc</b>	
<b>2</b>	$s_2$		$s_3$		<b>4</b>
<b>3</b>		$r_3$		$r_3$	
<b>4</b>		$s_5   r_2$		$r_2$	
<b>5</b>	$s_2$		$s_3$		<b>6</b>
<b>6</b>		$r_1$			

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

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

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

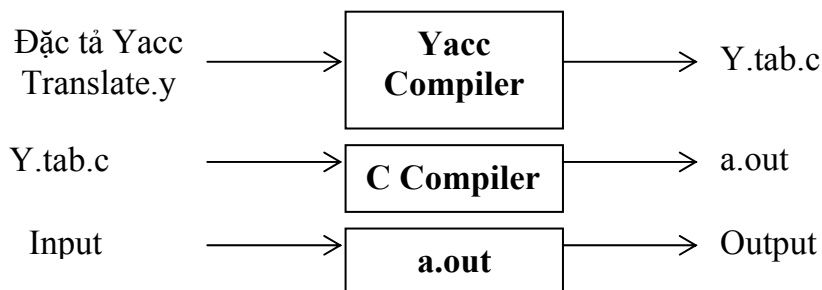
(if expr1 then if expr2 then  $a_1$  else  $a_2$ )

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

## VIII. BỘ SINH BỘ PHÂN TÍCH CÚ PHÁP

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

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



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

Một chương trình dịch có thể được xây dựng nhờ Yacc bằng phương thức được minh họa trong hình 4.18 trên. Trước tiên, cần chuẩn bị một tập tin, chẳng hạn là translate.y, chứa một đặc tả Yacc của chương trình dịch. Lệnh **yacc translate.y** của hệ UNIX sẽ biến đổi tập tin translate.y thành một chương trình C có tên là **y.tab.C** bằng phương pháp LALR đã trình bày ở trên. Chương trình **y.tab.C** là một biểu diễn của bộ phân tích cú pháp LALR được viết bằng ngôn ngữ C cùng với các thủ tục C khác có thể do người sử dụng chuẩn bị. Bằng cách dịch **y.tab.C** cùng với thư viện **ly** chứa chương trình phân tích cú pháp LR nhờ lệnh **cc y.tab.C - ly** chúng ta thu được một chương trình đối tượng **a.out** thực hiện quá trình dịch được đặc tả bởi chương trình Yacc ban đầu. Nếu cần thêm các thủ tục khác, chúng có thể được biên dịch hoặc được tải vào **y.tab.C** giống như mọi chương trình C khác.

## 2. Đặc tả YACC

Một chương trình nguồn Yacc bao gồm 3 phần:

### Phần khai báo

% %

### Các luật dịch

%%

### Các thủ tục

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

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

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

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

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

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

%%

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

%%

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

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

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

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

Mỗi luật sinh có dạng

$$\langle \text{vế trái} \rangle \rightarrow \langle \text{alt1} \rangle \mid \langle \text{alt2} \rangle \mid \dots \langle \text{altn} \rangle$$

được mô tả trong Yacc :

$$\begin{aligned} \langle \text{vế trái} \rangle &: \langle \text{alt1} \rangle \quad \{ \text{hành vi ngữ nghĩa 1} \} \\ &\mid \langle \text{alt2} \rangle \quad \{ \text{hành vi ngữ nghĩa 2} \} \\ &\dots \\ &\mid \langle \text{altn} \rangle \quad \{ \text{hành vi ngữ nghĩa n} \} \\ &; \end{aligned}$$

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

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

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

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

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

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

## BÀI TẬP CHƯƠNG IV

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

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

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

- a) Hãy chỉ ra các thành phần của văn phạm phi ngữ cảnh cho G.
- b) Viết văn phạm tương đương sau khi loại bỏ đệ quy trái.
- c) Xây dựng bộ phân tích cú pháp dự đoán cho văn phạm.
- d) Hãy dùng bộ phân tích cú pháp đã được xây dựng để vẽ cây phân tích cú pháp cho các câu nhập sau:

i) (a, a)

ii) (a, (a, a))

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

- e) Xây dựng dẫn xuất trái, dẫn xuất phải cho từng câu ở phần d
- f) Hãy cho biết ngôn ngữ do văn phạm G sinh ra ?

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

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

- a) Chứng minh văn phạm này là mơ hồ bằng cách xây dựng 2 chuỗi dẫn xuất trái khác nhau cho cùng câu nhập **abab**.
- b) Xây dựng các chuỗi dẫn xuất phải tương ứng cho câu nhập **abab**.
- c) Vẽ các cây phân tích cú pháp tương ứng.
- d) Văn phạm này sinh ra ngôn ngữ gì ?
- e) Xây dựng bộ phân tích cú pháp đệ quy lùi cho văn phạm trên. Có thể xây dựng bộ phân tích cú pháp dự đoán cho văn phạm này không ?

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

$$\text{bexpr} \rightarrow \text{bexpr } \mathbf{or} \text{ bterm} \mid \text{bterm}$$

$$\text{bterm} \rightarrow \text{bterm } \mathbf{and} \text{ bfactor} \mid \text{bfactor}$$

$$\text{bfactor} \rightarrow \mathbf{not} \text{ bfactor} \mid (\text{bexpr}) \mid \mathbf{true} \mid \mathbf{false}$$

- a) Hãy xây dựng bộ phân tích cú pháp dự đoán cho văn phạm G.
- b) Xây dựng cây phân tích cú pháp cho câu : **not ( true and false )**
- c) Chứng minh rằng văn phạm này sinh ra toàn bộ các biểu thức boole.



- d) Văn phạm G có là văn phạm mơ hồ không ? Tại sao ?
- e) Xây dựng bộ phân tích cú pháp SLR cho văn phạm.

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

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

- a) Chứng minh rằng văn phạm này sinh ra mọi biểu thức chính quy trên các ký hiệu a và b.
- b) Chứng tỏ đây là văn phạm mơ hồ.
- c) Xây dựng văn phạm không mơ hồ tương đương với thứ tự ưu tiên của các phép toán giảm dần như sau : phép bao đóng, phép nối kết, phép hợp.
- d) Vẽ cây phân tích cú pháp trong cả hai văn phạm trên cho câu nhập :  $a + b * c$
- e) Xây dựng bộ phân tích cú pháp dự đoán từ văn phạm không mơ hồ.
- f) Xây dựng bảng phân tích cú pháp SLR cho văn phạm G. Đề nghị một quy tắc giải quyết độ mơ hồ cho các biểu thức chính quy được phân tích một cách bình thường.

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

$$Stmt \rightarrow \text{if expr then stmt} \\ \mid \text{matched\_stmt}$$

$$Matched\_Stmt \rightarrow \text{if expr then matched\_stmt else stmt} \\ \mid \text{other}$$

Chứng minh rằng văn phạm này vẫn mơ hồ.

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

- a) Tập tất cả các chuỗi 0 và 1 sao cho mỗi số 0 có ít nhất một số 1 ở ngay sau nó.
- b) Các chuỗi 0 và 1 với số số 0 bằng số số 1.
- c) Các chuỗi 0 và 1 với số số 0 không bằng số số 1.
- d) Các chuỗi 0 và 1 không chứa chuỗi 001 như chuỗi con.

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

$$S \rightarrow aSa \mid aa$$

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

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

$$S \rightarrow \mathbf{aAB}$$

$$A \rightarrow \mathbf{Abc} \mid \mathbf{b}$$

$$B \rightarrow \mathbf{d}$$

- a) Xây dựng bộ phân tích cú pháp dự đoán cho văn phạm .
- b) Hãy dùng bộ phân tích cú pháp đã được xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **abccd**

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

$$E \rightarrow E \textbf{ or } T \mid T$$

$$T \rightarrow T \textbf{ and } F \mid F$$

$$F \rightarrow ( E ) \mid \textbf{ not } F \mid \textbf{ id}$$

- a) Hãy xây dựng bộ phân tích cú pháp dự đoán cho văn phạm.
- b) Vẽ cây phân tích cú pháp cho câu nhập : id and not ( **id or id** )

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

$$S \rightarrow AB$$

$$A \rightarrow \mathbf{Ab} \mid \mathbf{a}$$

$$B \rightarrow \mathbf{cB} \mid \mathbf{d}$$

- a) Xây dựng bộ phân tích cú pháp thứ tự ưu tiên cho văn phạm .
- b) Hãy dùng bộ phân tích cú pháp đã xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **abccd**

**4.11.** Cho văn phạm G:

$$S \rightarrow D \bullet D \mid D$$

$$D \rightarrow DB \mid B$$

$$B \rightarrow \mathbf{0} \mid \mathbf{1}$$

- a) Xây dựng bộ phân tích cú pháp thứ tự ưu tiên cho văn phạm .
- b) Hãy dùng bộ phân tích cú pháp đã xây dựng để phát sinh cây phân tích cú pháp cho câu nhập: **101•101**

**4.12.** Cho văn phạm G

$$\text{Assign} \rightarrow \mathbf{id} := \mathbf{exp}$$

$$\text{Exp} \rightarrow \text{Exp} + \text{Term} \mid \text{Term}$$

$$\text{Term} \rightarrow \text{Term} * \text{Factor} \mid \text{Factor}$$

$$\text{Factor} \rightarrow \mathbf{id} \mid ( \text{Exp} )$$

- a) Xây dựng bộ phân tích cú pháp thứ tự ưu tiên cho văn phạm .
- b) Hãy dùng bộ phân tích cú pháp đã được xây dựng để phát sinh cây phân tích cú pháp cho câu nhập:  $\mathbf{id := id + id * id}$

**4.13.** Cho văn phạm mơ hồ như sau:

$$S \rightarrow AS \mid \mathbf{b}$$

$$A \rightarrow SA \mid \mathbf{a}$$

- a) Xây dựng họ tập hợp mục LR(0) cho văn phạm này.
- b) Xây dựng bảng phân tích cú pháp SLR .
- c) Thực hiện quá trình phân tích cú pháp SLR khả triển cho chuỗi nhập :  $\mathbf{abab}$
- d) Xây dựng bảng phân tích cú pháp chính tắc .
- e) Xây dựng bảng phân tích cú pháp LALR .

**4.14.** Cho văn phạm G như sau:

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

$$T \rightarrow TF \mid F$$

$$F \rightarrow F * \mid \mathbf{a} \mid \mathbf{b}$$

- a) Xây dựng bảng phân tích cú pháp SLR cho văn phạm này.
- b) Thực hiện quá trình phân tích cú pháp SLR cho chuỗi nhập :  $\mathbf{b + ab * a}$
- c) Xây dựng bảng phân tích cú pháp LALR.

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

$$S \rightarrow \mathbf{Aa} \mid \mathbf{bAc} \mid \mathbf{dc} \mid \mathbf{bda}$$

$$A \rightarrow \mathbf{d}$$

là LALR(1) nhưng không phải SLR(1).

**4.16.** Cho văn phạm G như sau:

$$E \rightarrow E \mathbf{sub} R \mid E \mathbf{sup} E \mid \{ E \} \mid \mathbf{c}$$

$$R \rightarrow E \mathbf{sup} E \mid E$$

- a) Xây dựng bảng phân tích cú pháp SLR cho văn phạm này.
- b) Đề nghị một quy tắc giải quyết độ để các biểu thức text có thể được phân tích một cách bình thường.

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

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

## CHƯƠNG V

### DỊCH TRỰC TIẾP CÚ PHÁP

#### Nội dung chính:

Khi viết một chương trình bằng một ngôn ngữ lập trình nào đó, ngoài việc quan tâm đến cấu trúc của chương trình (cú pháp – văn phạm), ta còn phải chú ý đến ý nghĩa của chương trình. Như vậy, khi thiết kế một trình biên dịch, ta không những chú ý đến văn phạm mà còn chú ý đến cả ngữ nghĩa. Chương 5 trình bày các cách biểu diễn ngữ nghĩa của một chương trình. Mỗi ký hiệu văn phạm kết hợp với một tập các *thuộc tính* – các thông tin. Mỗi luật sinh kết hợp với một tập các *luật ngữ nghĩa* – các quy tắc xác định trị của các thuộc tính. Việc đánh giá các luật ngữ nghĩa được sử dụng để thực hiện một công việc nào đó như tạo ra mã trung gian, lưu thông tin vào bảng ký hiệu, xuất các thông báo lỗi, v.v. Ta sẽ thấy rõ việc đánh giá này ở các chương sau: 6, 8, 9. Hai cách để kết hợp các luật sinh với các luật ngữ nghĩa được trình bày trong chương là: *Định nghĩa trực tiếp cú pháp* và *Lược đồ dịch*. Ở mức quan niệm, bằng cách sử dụng định nghĩa trực tiếp cú pháp hoặc lược đồ dịch, ta phân tích dòng thẻ từ, xây dựng cây phân tích cú pháp và duyệt cây khi cần để đánh giá các luật ngữ nghĩa tại các nút của cây.

#### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải nắm được:

- Các cách kết hợp các luật sinh với các luật ngữ nghĩa: Định nghĩa trực tiếp cú pháp và Lược đồ dịch.
- Biết cách thiết kế chương trình – bộ dịch dự đoán - thực hiện một công việc nào đó từ một lược đồ dịch hay từ một định nghĩa trực tiếp cú pháp xác định.

#### Tài liệu tham khảo:

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

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

### I. ĐỊNH NGHĨA TRỰC TIẾP CÚ PHÁP

Định nghĩa trực tiếp cú pháp là sự tổng quát hóa một văn phạm phi ngữ cảnh, trong đó mỗi ký hiệu văn phạm kết hợp với một tập các thuộc tính.

Cây phân tích cú pháp có trình bày giá trị các thuộc tính tại mỗi nút gọi là cây chú thích .

#### 1. Khái niệm về định nghĩa trực tiếp cú pháp

Trong một định nghĩa trực tiếp cú pháp, mỗi luật sinh  $A \rightarrow \alpha$  kết hợp một tập luật ngữ nghĩa có dạng  $\mathbf{b} := \mathbf{f}(\mathbf{c1}, \mathbf{c2}, \dots, \mathbf{ck})$  trong đó  $\mathbf{f}$  là một hàm và :

- 1-  $b$  là một thuộc tính tổng hợp của  $A$  và  $c_1, c_2, \dots, c_k$  là các thuộc tính của các ký hiệu văn phạm của luật sinh. Hoặc
- 2-  $b$  là một thuộc tính kế thừa của một trong các ký hiệu văn phạm trong vế phải của luật sinh và  $c_1, c_2, \dots, c_k$  là các thuộc tính của các ký hiệu văn phạm của luật sinh.

Ta nói  $b$  phụ thuộc  $c_1, c_2, \dots, c_k$ .

### 1. Thuộc tính tổng hợp

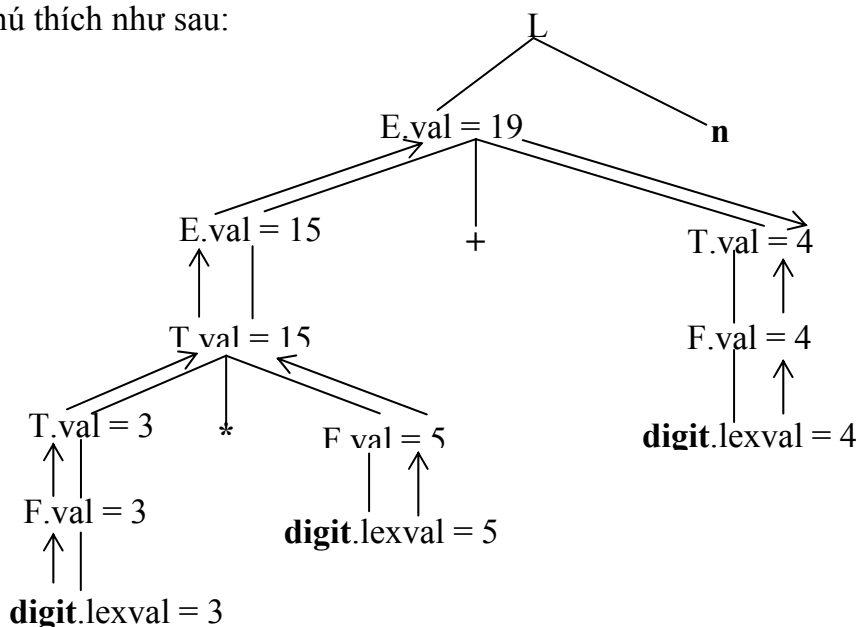
- Là thuộc tính mà giá trị của nó tại mỗi nút trên cây phân tích cú pháp được tính từ giá trị thuộc tính tại các nút con của nó.
- Định nghĩa trực tiếp cú pháp chỉ sử dụng các thuộc tính tổng hợp gọi là định nghĩa  $S\_thuộc\ tính$ .
- Cây phân tích cú pháp của định nghĩa  $S\_thuộc\ tính$  có thể được chú thích từ dưới lên trên.

**Ví dụ 5.1:** Xét định nghĩa trực tiếp cú pháp

Luật sinh	Luật ngữ nghĩa
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

**Hình 5.1** - Định nghĩa trực tiếp cú pháp cho một máy tính tay đơn giản

Định nghĩa này kết hợp một thuộc tính tổng hợp có giá trị nguyên  $val$  với từng ký hiệu chưa kết thúc  $E, T$  và  $F$ . Token  $digit$  có một thuộc tính tổng hợp  $lexval$  với giả sử rằng giá trị của thuộc tính này được cung cấp bởi bộ phân tích từ vựng. Đây là một định nghĩa  $S\_thuộc\ tính$ . Với biểu thức  $3 * 5 + 4n$  ( $n$  là ký hiệu newline) có cây chú thích như sau:



**Hình 5.2-** Cây chú thích cho biểu thức  $3 * 5 + 4n$

## 2. Thuộc tính kế thừa

- Là một thuộc tính mà giá trị của nó được xác định từ giá trị các thuộc tính của các nút cha hoặc anh em của nó.
- Nói chung ta có thể viết một định nghĩa trực tiếp cú pháp thành một định nghĩa S\_ thuộc tính. Nhưng trong một số trường hợp, việc sử dụng thuộc tính kế thừa lại thuận tiện vì tính tự nhiên của nó.

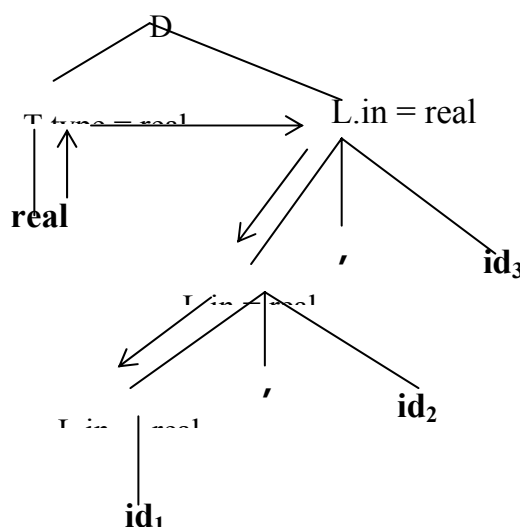
**Ví dụ 5.2:** Xét định nghĩa trực tiếp cú pháp sau cho sự khai báo kiểu cho biến:

Luật sinh	Luật ngữ nghĩa
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := integer$
$T \rightarrow \text{real}$	$T.type := real$
$L \rightarrow L_1, id$	$L_1.in := L.in; addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

**Hình 5.3** - Định nghĩa trực tiếp cú pháp với thuộc tính kế thừa  $L.in$

$type$  là thuộc tính tổng hợp kết hợp với ký hiệu chưa kết thúc  $T$ , giá trị của nó được xác định bởi từ khóa trong khai báo. Bằng cách sử dụng thuộc tính kế thừa  $in$  kết hợp với ký hiệu chưa kết thúc  $L$  chúng ta xác định được kiểu cho các danh biểu và dùng thủ tục  $addtype$  đưa kiểu này vào trong bảng ký hiệu tương ứng với danh biểu.

**Ví dụ 5.3:** Xét phép khai báo: **real id1, id2, id3**. Ta có cây chú thích:



**Hình 5.4-** Cây phân tích cú pháp với thuộc tính kế thừa  $in$  tại mỗi nút được gán nhãn  $L$

- Đồ thị phụ thuộc là một đồ thị có hướng mô tả sự phụ thuộc giữa các thuộc tính tại mỗi nút của cây phân tích cú pháp.
- Cho một cây phân tích cú pháp thì đồ thị phụ thuộc tương ứng được xây dựng theo giải thuật sau:

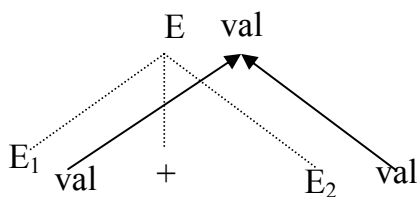
FOR với mỗi một thuộc tính  $a$  của ký hiệu văn phạm tại  $n$  DO

FOR với mỗi một nút  $n$  trên cây phân tích cú pháp DO

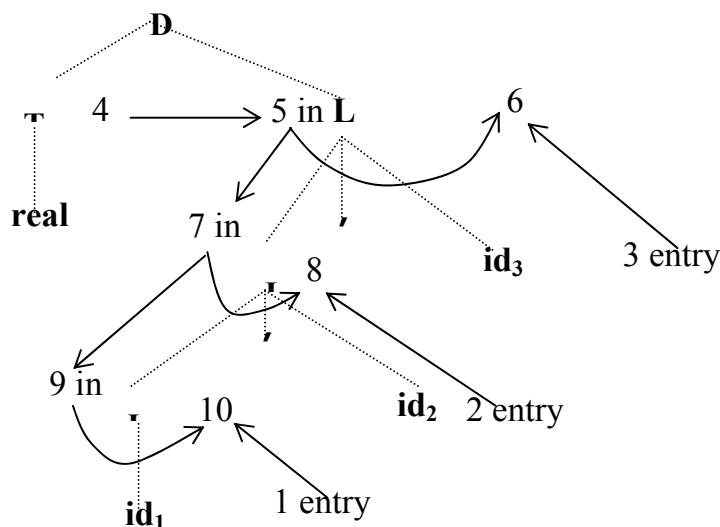
FOR i:=1 TO k DO

**Ví dụ 5.4:** Với định nghĩa S thuộc tính

Ta có đồ thị phụ thuộc:



**Ví dụ 5.5:** Dựa vào định nghĩa trực tiếp cú pháp trong ví dụ 5.2, ta có đồ thị phụ thuộc của khai báo **real id1, id2, id3**



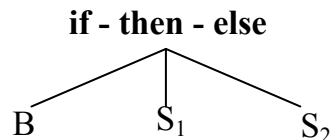
119



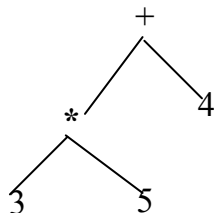
**Chú ý:** Với luật ngữ nghĩa dạng  $b = f(c_1, c_2, \dots, c_k)$  có chứa lời gọi thủ tục thì chúng ta tạo ra một thuộc tính tổng hợp giả. Trong ví dụ của chúng ta là nút 6, 8, 10.

## II. XÂY DỰNG CÂY CÚ PHÁP

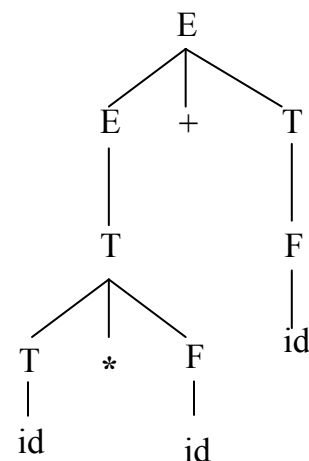
- Cây cú pháp (syntax - tree) là dạng rút gọn của cây phân tích cú pháp dùng để biểu diễn cấu trúc ngôn ngữ.
- Trong cây cú pháp các toán tử và từ khóa không phải là nút lá mà là các nút trong. Ví dụ với luật sinh  $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$  được biểu diễn bởi cây cú pháp:



- Một kiểu rút gọn khác của cây cú pháp là chuỗi các luật sinh đơn được rút gọn lại. Chẳng hạn ta có:



được rút gọn từ



### 1. Xây dựng cây cú pháp cho biểu thức

Tương tự như việc dịch một biểu thức thành dạng hậu tố.

Xây dựng cây con cho biểu thức con bằng cách tạo ra một nút cho toán hạng và toán tử.

Con của nút toán tử là gốc của cây con biểu diễn cho biểu thức con toán hạng của toán tử đó.

Mỗi một nút có thể cài đặt bằng một mẫu tin có nhiều trường.

Trong nút toán tử, có một trường chỉ toán tử như là nhãn của nút, các trường còn lại chứa con trái, trỏ tới các nút toán hạng.

Để xây dựng cây cú pháp cho biểu thức chúng ta sử dụng các hàm sau đây:

- mknode(op, left, right):** Tạo một nút toán tử có nhãn là op và hai trường chứa con trái, trỏ tới con trái left và con phải right.

2. **mkleaf(id, entry)**: Tạo một nút lá với nhãn là id và một trường chứa con trỏ entry, trỏ tới ô trong bảng ký hiệu.

3. **mkleaf(num, val)**: Tạo một nút lá với nhãn là num và trường val, giá trị của số.

**Ví dụ 5.6:** Để xây dựng cây cú pháp cho biểu thức: **a - 4 + c** ta dùng một dãy các lời gọi các hàm nói trên.

(1): p1 := mkleaf(id, entrya)

(4): p4 := mkleaf(id, entryc)

(2): p2 := mkleaf(num, 4)

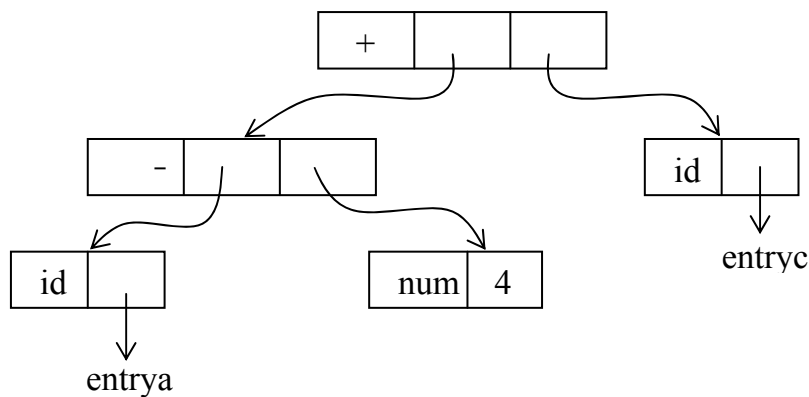
(5): p5 := mknnode('+', p3, p4)

(3): p3 := mknnode('-', p1, p2)

Cây được xây dựng từ dưới lên

entrya là con trỏ, trỏ tới ô của a trong bảng ký hiệu

entryc là con trỏ, trỏ tới ô của c trong bảng ký hiệu



**Hình 5.7-** Cây cú pháp cho biểu thức  $a - 4 + c$

## 2. Xây dựng cây cú pháp từ định nghĩa trực tiếp cú pháp

Căn cứ vào các luật sinh văn phạm và luật ngữ nghĩa kết hợp mà ta phân bổ việc gọi các hàm mknnode và mkleaf để tạo ra cây cú pháp.

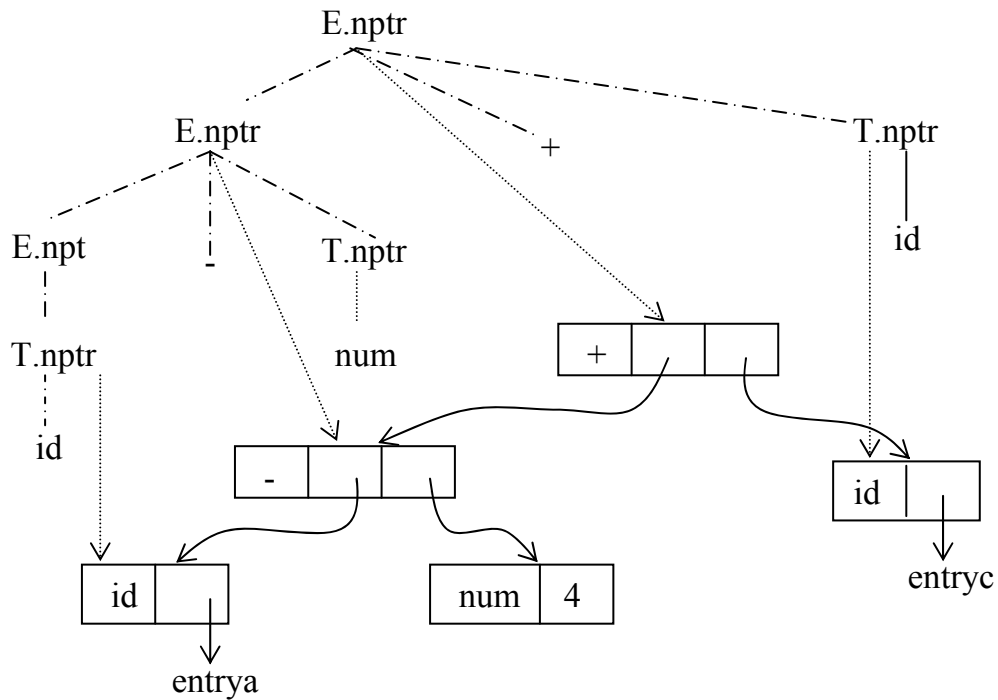
**Ví dụ 5.7:** Định nghĩa trực tiếp cú pháp giúp việc xây dựng cây cú pháp cho biểu thức là:

Luật sinh	Luật ngữ nghĩa
$E \rightarrow E_1 + T$	$E.nptr := mknnode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknnode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow id$	$T.nptr := mkleaf(id, id.entry)$
$T \rightarrow num$	$T.nptr := mkleaf(num, num.val)$

**Hình 5.8 -** Định nghĩa trực tiếp cú pháp để tạo cây cú pháp cho biểu thức

Các nút trên cây phân tích cú pháp có nhãn là các ký hiệu chưa kết thúc E và T sử dụng thuộc tính tổng hợp nptr để lưu con trỏ trỏ tới một nút trên cây cú pháp.

Với biểu thức  $a - 4 + c$  ta có cây phân tích cú pháp (biểu diễn bởi đường chấm) trong hình 5.9.



**Hình 5.9 - Xây dựng cây cú pháp cho  $a - 4 + c$**

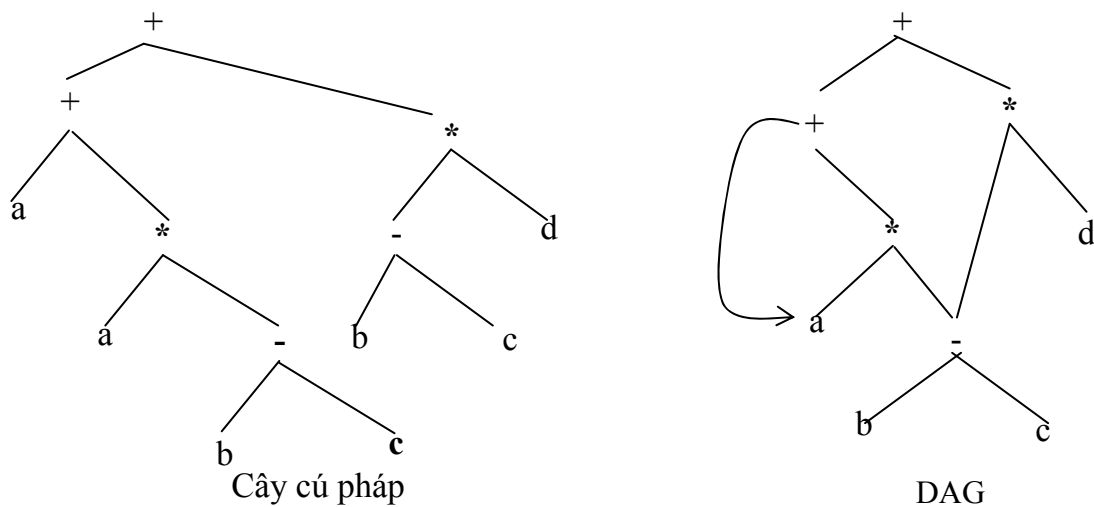
Luật ngữ nghĩa cho phép tạo ra cây cú pháp.

Cây cú pháp có ý nghĩa về mặt cài đặt còn cây phân tích cú pháp chỉ có ý nghĩa về mặt logic.

### 3. Đồ thị có hướng không tuần hoàn cho biểu thức (Directed Acyclic Graph - DAG)

DAG cũng giống như cây cú pháp, tuy nhiên trong cây cú pháp các biểu thức con giống nhau được biểu diễn lặp lại còn trong DAG thì không. Trong DAG, một nút con có thể có nhiều “cha”.

**Ví dụ 5.8:** Cho biểu thức  $a + a * (b - c) + (b - c) * d$ . Ta có cây cú pháp và DAG:



**Hình 5.10 - Cây cú pháp và DAG của một biểu thức**

Để xây dựng một DAG, trước khi tạo một nút phải kiểm tra xem nút đó đã tồn tại chưa, nếu đã tồn tại thì hàm tạo nút (mknode, mkleaf) trả về con trỏ của nút đã tồn tại, nếu chưa thì tạo nút mới.

### Cài đặt DAG:

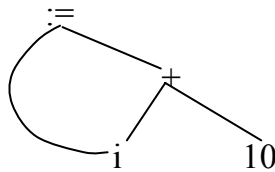
Người ta thường sử dụng một mảng mẫu tin, mỗi mẫu tin là một nút. Ta có thể tham khảo tới nút bằng chỉ số của mảng.

#### Ví dụ 5.9:

Lệnh gán

$i := i + 10$

DAG



Biểu diễn

1	id	entry i	
2	num	10	
3	+	1	2
4	:=	1	3

Hình 5.11- Minh họa cài đặt DAG

Nút 1: có nhãn là id, con trỏ trỏ tới entry i.

Nút 2: có nhãn là num, giá trị là 10.

Nút 3: có nhãn là +, con trái là nút 1, con phải là nút 2.

Nút 4: có nhãn là :=, con trái là nút 1, con phải là nút 3.

**Giải thuật: Phương pháp số giá trị (value – number) để xây dựng một nút trong DAG.**

Giả sử rằng các nút được lưu trong một mảng và mỗi nút được tham khảo bởi số giá trị của nó. Mỗi một nút toán tử là một bộ ba  $\langle op, l, r \rangle$

**Input:** Nhãn op, nút l và nút r.

**Output:** Một nút với  $\langle op, l, r \rangle$

**Phương pháp:** Tìm trong mảng một nút m có nhãn là op con trái là l, con phải là r. Nếu tìm thấy thì trả về m, ngược lại tạo ra một nút mới n, có nhãn là op, con trái là l, con phải là r và trả về n.

### III. ĐÁNH GIÁ DƯỚI LÊN ĐỐI VỚI ĐỊNH NGHĨA S\_THUỘC TÍNH

#### 1. Sử dụng Stack

Như đã biết, định nghĩa S\_ thuộc tính chỉ chứa các thuộc tính tổng hợp do đó phương pháp phân tích dưới lên là phù hợp với định nghĩa trực tiếp cú pháp này. Phương pháp phân tích dưới lên sử dụng một STACK để lưu trữ thông tin về cây con đã được phân tích. Chúng ta có thể mở rộng STACK này để lưu trữ giá trị thuộc tính tổng hợp. STACK được cài đặt bởi một cặp mảng state và val.

Giả sử luật ngữ nghĩa  $A.a := f(X.x, Y.y, Z.z)$  kết hợp với luật sinh  $A \rightarrow XYZ$ . Trước khi XYZ được rút gọn thành A thì  $val[top] = Z.z$ ,  $val[top - 1] = Y.y$ ,  $val[top - 2]$

= X.x. Sau khi rút gọn, top bị giảm 2 đơn vị, A nằm trong state[top] và thuộc tính tổng hợp nằm trong val[top].

State	val
...	...
X	X.x
Y	Y.y
Z	Z.z

←  
top

Mỗi ô trong stack là một con trỏ trỏ tới bảng phân tích LR(1). Nếu phần tử thứ i của stack là ký hiệu A thì val[i] là giá trị thuộc tính kết hợp với A.

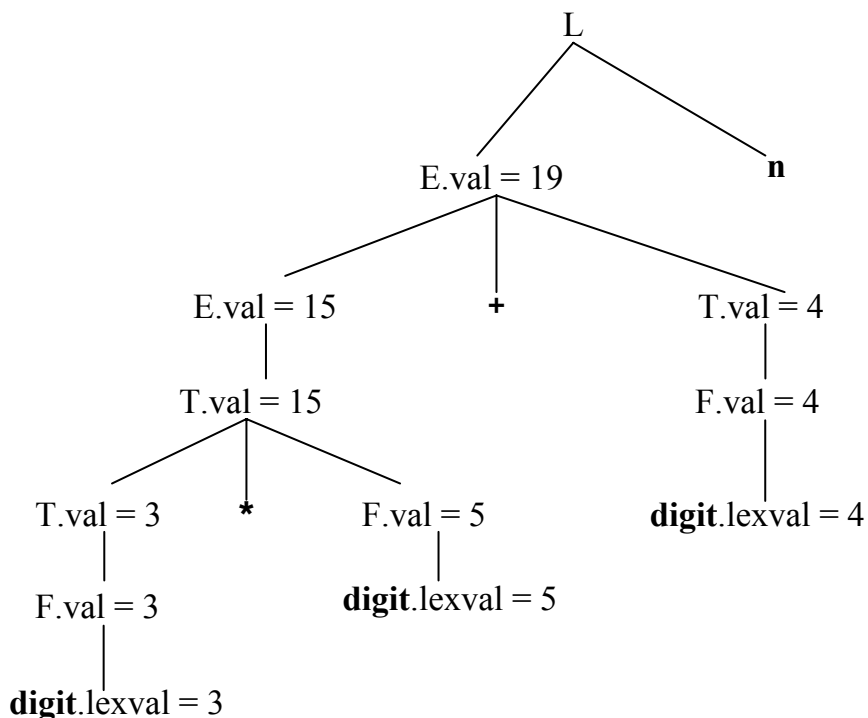
**Hình 5.12** - Stack phân tích cú pháp vào một trường lưu giữ thuộc tính tổng hợp

## 2. Ví dụ

**Ví dụ 5.10:** Xét định nghĩa trực tiếp cú pháp:

Luật sinh	Luật ngữ nghĩa
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$

Với biểu thức  $3 * 5 + 4$  n ta có cây chú thích:



**Hình 5.13** – Cây chú thích cho biểu thức  $3 * 5 + 4 n$

Cây chú thích này có thể được đánh giá bằng một bộ phân tích cú pháp LR từ dưới lên trên. Chú ý rằng bộ phân tích đã nhận biết giá trị thuộc tính `digit.lexval`. Khi `digit` được đưa vào stack thì token `digit` được đưa vào `state[top]` và giá trị thuộc tính của nó được đưa vào `val[top]`. Chúng ta có thể sử dụng kỹ thuật trong mục VI của chương IV để xây dựng bộ phân tích LR. Để đánh giá các thuộc tính chúng ta thay đổi bộ phân tích cú pháp để thực hiện đoạn mã sau:

Luật sinh	Luật ngữ nghĩa
$L \rightarrow En$	<code>print(val[top])</code>
$E \rightarrow E_1 + T$	<code>val[ntop] := val[top - 2] + val[top]</code>
$E \rightarrow T$	
$T \rightarrow T_1 * F$	<code>val[ntop] := val[top - 2] * val[top]</code>
$T \rightarrow F$	
$F \rightarrow (E)$	<code>val[ntop] := val[top - 1]</code>
$F \rightarrow \text{digit}$	

**Hình 5.14-** Cài đặt một máy tính tay sử dụng bộ phân tích cú pháp LR

Khi một luật sinh với  $r$  ký hiệu bên vế phải được rút gọn thì  $ntop = top - r + 1$ . Sau khi một đoạn mã thực hiện thì đặt  $top = ntop$

Bảng sau trình bày quá trình thực hiện của bộ phân tích cú pháp

Input	State	Val	Luật sinh được dùng
3 * 5 + 4 n	—	—	
* 5 + 4 n	3	3	
* 5 + 4 n	F	3	$F \rightarrow \text{digit}$
* 5 + 4 n	T	3	$T \rightarrow F$
5 + 4 n	T*	3 -	
+ 4 n	T * 5	3 - 5	
+ 4 n	T * F	3 - 5	$F \rightarrow \text{digit}$
+ 4 n	T	15	$T \rightarrow T * F$
+ 4 n	E	15	$E \rightarrow T$
4 n	E +	15 -	
n	E + 4	15 - 4	
n	E + F	15 - 4	$F \rightarrow \text{digit}$
n	E + T	15 - 4	$T \rightarrow F$

n	E	19	$E \rightarrow E + T$
	E n	19 -	
	L	19	$L \rightarrow En$

**Hình 5.15-** Các phép chuyển được tạo ra bởi bộ thông dịch trên chuỗi nhập  $3 * 5 + 4n$

#### IV. ĐỊNH NGHĨA L THUỘC TÍNH

##### 1. Định nghĩa L\_thuộc tính.

Mỗi định nghĩa trực tiếp cú pháp là một định nghĩa L thuộc tính nếu mỗi một thuộc tính kế thừa của  $X_i$  ( $1 \leq i \leq n$ ) trong vế phải của luật sinh  $A \rightarrow X_1 X_2 \dots X_n$  phụ thuộc chỉ vào:

1. Các thuộc tính của  $X_1, X_2, \dots, X_{i-1}$
2. Các thuộc tính kế thừa của A.

**Ví dụ 5.11:** Cho định nghĩa trực tiếp cú pháp:

Luật sinh	Luật ngữ nghĩa
$A \rightarrow L M$	$L.i := l(A.i)$ $M.i := m(L.s) \quad A.s := f(M.s)$ $R.i := r(A.i)$
$A \rightarrow Q R$	$Q.i := q(R.s) \quad A.s := f(Q.r)$

**Hình 5.16 -** Định nghĩa trực tiếp cú pháp không phải là định nghĩa L\_thuộc tính

Đây không phải là một định nghĩa L\_thuộc tính vì thuộc tính kế thừa  $Q.i$  phụ thuộc vào thuộc tính  $R.s$  của ký hiệu bên phải nó trong luật sinh.

##### 2. Lược đồ dịch

Lược đồ dịch là một văn phạm phi ngữ cảnh trong đó các thuộc tính được kết hợp với các ký hiệu văn phạm và các hành vi ngữ nghĩa nằm trong cặp dấu  $\{ \}$  được xen vào bên phải của luật sinh.

**Ví dụ 5.12:** Lược đồ dịch một biểu thức trung tố với phép cộng và trừ thành dạng hậu tố:

$E \rightarrow T R$

$R \rightarrow \text{addop } T \{ \text{print} ( \text{addop.lexeme} ) \} R_1 \mid \varepsilon$

$T \rightarrow \text{num} \quad \{ \text{print} ( \text{num.val} ) \}$

Với biểu thức  $9 - 5 + 2$  ta có cây phân tích cú pháp (hình 5.16)

Để xây dựng một lược đồ dịch, chúng ta xét hai trường hợp sau đây:

**Trường hợp 1:** Chỉ chứa thuộc tính tổng hợp:



Với mỗi một luật ngữ nghĩa, ta tạo một hành vi ngữ nghĩa và đặt hành vi này vào cuối vế phải luật sinh.

### Ví dụ 5.13:

Luật sinh

$$T \rightarrow T_1 * F$$

Luật ngữ nghĩa

$$T.val := T_1.val * F.val$$

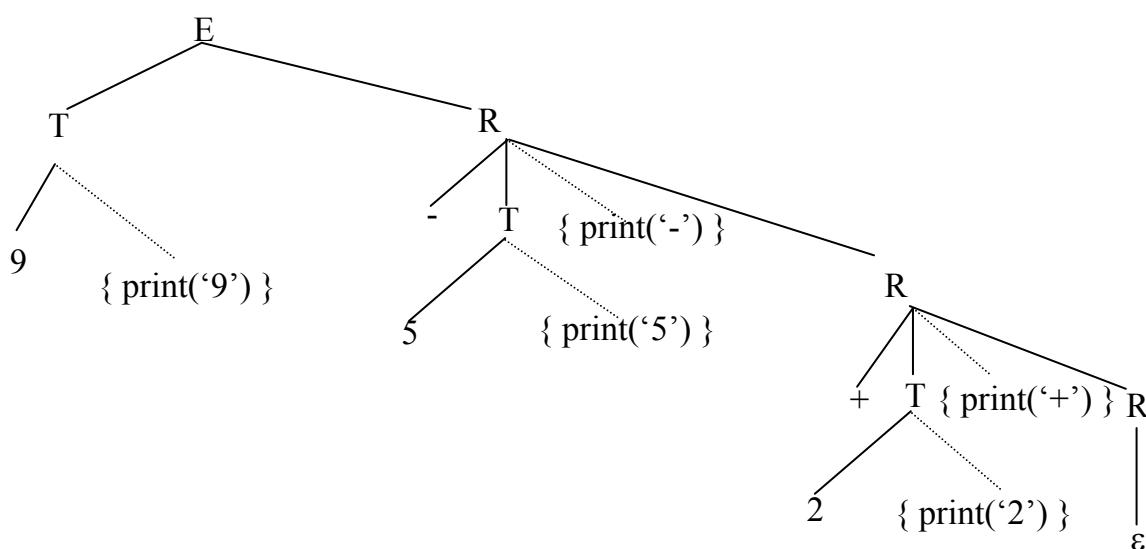
Ta có lược đồ dịch:

$$T \rightarrow T_1 * F \{ T.val := T_1.val * F.val \}$$

**Trường hợp 2:** Có cả thuộc tính tổng hợp và kế thừa phải thỏa mãn 3 yêu cầu sau đây:

1. Thuộc tính kế thừa của một ký hiệu trong vế phải của luật sinh phải được xác định trong hành vi nằm trước ký hiệu đó.
2. Một hành vi không được tham khảo tới thuộc tính tổng hợp của một ký hiệu nằm bên phải hành vi đó.
3. Thuộc tính tổng hợp của ký hiệu chưa kết thúc trong vế trái chỉ có thể được xác định sau khi tất cả các thuộc tính mà nó tham khảo đã được xác định. Hành vi xác định các thuộc tính này luôn đặt cuối vế phải của luật sinh.

Với một định nghĩa trực tiếp cú pháp  $L\_thuộc\ tính$  ta có thể xây dựng lược đồ dịch thỏa mãn 3 yêu cầu nói trên.



**Hình 5.17 - Cây phân tích cú pháp của các hoạt động biểu diễn 9-5+2**

**Ví dụ 5.14:** Bộ xử lý các công thức toán học – EQN - có thể xây dựng các biểu thức toán học từ các toán tử sub (subscripts) và sup (superscripts). Chẳng hạn:

**input**

BOX **sub** box

a **sub** {i **sup** 2 }

**output**

BOX<sub>box</sub>

$a_{i^2}$

Để xác định chiều rộng và chiều cao của các hộp ta có định nghĩa  $L\_thuộc$  tính như sau:

Luật sinh	Luật ngữ nghĩa
$S \rightarrow B$	$B.ps := 10$ $S.ht := B.ht$
$B \rightarrow B_1 B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := \max(B_1.ht, B_2.ht)$
$B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps := B.ps$ $B_2.ps := \text{shrink}(B.ps)$ $B.ht := \text{disp}(B_1.ht, B_2.ht)$
$B \rightarrow \text{text}$	$B.ht := \text{text.h} * B.ps$

**Hình 5.18** - Định nghĩa trực tiếp cú pháp xác định kích thước và chiều cao của các hộp

Trong đó:

- Ký hiệu chưa kết thúc  $B$  biểu diễn một công thức.
- Luật sinh  $B \rightarrow BB$  biểu diễn sự kề nhau của hai hộp.
- Luật sinh  $B \rightarrow B \text{ sub } B$  biểu diễn sự sắp đặt, trong đó hộp chỉ số thứ 2 có kích thước nhỏ hơn, nằm thấp hơn hộp thứ nhất.
- Thuộc tính kế thừa  $ps$  (point size - kích thước điểm) phản ánh độ lớn của công thức.
- Luật sinh  $B \rightarrow \text{text}$  ứng với luật ngữ nghĩa  $B.ht := \text{text.h} * B.ps$  lấy chiều cao thực của  $\text{text}$  ( $h$ ) nhân với kích thước điểm của  $B$  để có được chiều cao của hộp.
- Luật sinh  $B \rightarrow B_1 B_2$  được áp dụng thì  $B_1, B_2$  kế thừa kích thước điểm của  $B$  bằng luật copy. Độ cao của  $B$  bằng giá trị lớn nhất trong độ cao của  $B_1, B_2$ .
- Khi luật sinh  $B \rightarrow B_1 \text{ sub } B_2$  được áp dụng thì hàm  $\text{shrink}$  sẽ giảm kích thước điểm của  $B_2$  còn 30% và hàm  $\text{disp}$  đẩy hộp  $B_2$  xuống.

Đây là một định nghĩa  $L\_thuộc$  tính vì chỉ có duy nhất một thuộc tính kế thừa  $ps$  và thuộc tính này chỉ phụ thuộc vào vế trái của luật sinh.

Dựa vào 3 yêu cầu nói trên, ta xen các hành vi ngữ nghĩa tương ứng với luật ngữ nghĩa vào vế phải của mỗi luật sinh để được lược đồ dịch.

$S \rightarrow$	$\{B.ps := 10\}$
$B$	$\{S.ht := B.ht\}$
$B \rightarrow$	$\{B_1.ps := B.ps\}$
$B_1$	$\{B_2.ps := B.ps\}$
$B_2$	$\{B.ht := \max(B_1.ht, B_2.ht)\}$

$$\begin{array}{ll}
B \rightarrow & \{B_1.ps := B.ps\} \\
B_1 & \\
\text{sub} & \{B_2.ps := shrink(B.ps)\} \\
B_2 & \{B.ht := disp(B_1.ht, B_2.ht)\} \\
B \rightarrow \text{text} & \{B.ht := text.h * B.ps\}
\end{array}$$

**Hình 5.19** - Lược đồ dịch được tạo ra từ hình 5.18

**Chú ý:** Để dễ đọc mỗi ký hiệu văn phạm trong luật sinh được viết trên một dòng và hành vi được viết vào bên phải.

Chẳng hạn:

$$\begin{array}{ll}
S \rightarrow \{B.ps := 10\} & B \{S.ht := B.ht\} \\
\text{Được viết thành} & S \rightarrow \{B.ps := 10\} \\
& B \{S.ht := B.ht\}
\end{array}$$

## V. DỊCH TRÊN XUỐNG

### 1. Loại bỏ đệ quy trái

Vấn đề loại bỏ đệ quy trái của một văn phạm đã được trình bày trong mục III của chương IV. Ở đây chúng ta giải quyết vấn đề chuyển một lược đồ dịch của văn phạm đệ quy trái thành một lược đồ dịch mới không còn đệ quy.

Giả sử, ta có lược đồ dịch dạng

$$\begin{array}{ll}
A \rightarrow A_1 Y & \{A.a := g(A_1.a, Y.y)\} \\
A \rightarrow X & \{A.a := f(X.x)\}
\end{array}$$

Đây là một văn phạm đệ quy trái, áp dụng giải thuật khử đệ quy trái ta được văn phạm không đệ quy trái

$$\begin{array}{ll}
A \rightarrow X R \\
R \rightarrow Y R \mid \varepsilon
\end{array}$$

Bổ sung hành vi ngữ nghĩa cho văn phạm ta được lược đồ dịch:

$$\begin{array}{ll}
A \rightarrow X & \{R.i := f(X.x)\} \\
R & \{A.a := R.s\} \\
R \rightarrow Y & \{R_1.i := g(R.i, Y.y)\} \\
R_1 & \{R.s := R_1.s\} \\
R \rightarrow \varepsilon & \{R.s := R.i\}
\end{array}$$

**Ví dụ 5.15:** Xét lược đồ dịch của văn phạm đệ quy trái cho biểu thức.

$$\begin{array}{ll}
E \rightarrow E_1 + T & \{E.val := E_1.val + T.val\} \\
E \rightarrow E_1 - T & \{E.val := E_1.val - T.val\} \\
E \rightarrow T & \{E.val := T.val\}
\end{array}$$

$$\begin{aligned}
T &\rightarrow (E) && \{T.val := E.val\} \\
T &\rightarrow \text{num} && \{T.val := num.val\}
\end{aligned}$$

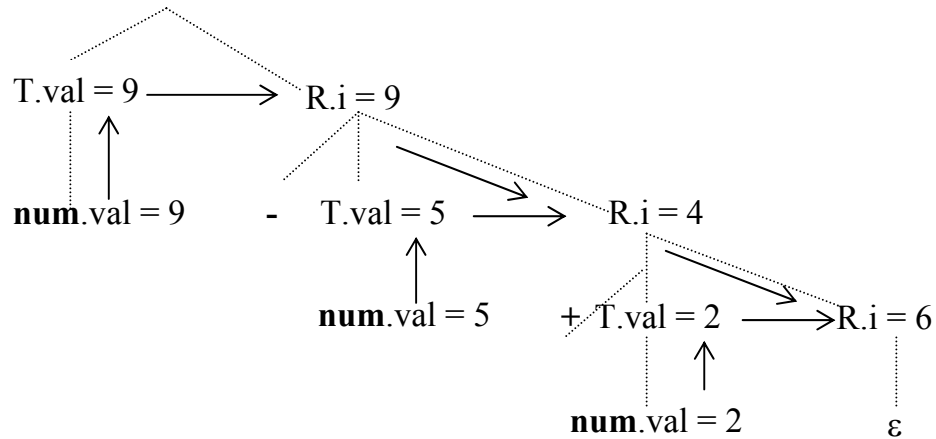
**Hình 5.20** - Lược đồ dịch của một văn phạm đệ quy trái

Vận dụng ý kiến trên ta khử đệ quy trái để được lược đồ dịch không đệ quy trái

$$\begin{aligned}
E &\rightarrow T \quad \{R.i := T.val\} \\
R &\quad \{E.val := R.s\} \\
R &\rightarrow + \\
&\quad T \quad \{R_l.i := R.i + T.val\} \\
&\quad R_l \quad \{R.s := R_l.s\} \\
R &\rightarrow - \\
&\quad T \quad \{R_l.i := R.i - T.val\} \\
&\quad R_l \quad \{R.s := R_l.s\} \\
R &\rightarrow \varepsilon \quad \{R.s := R.i\} \\
T &\rightarrow ( \\
&\quad E \\
&\quad ) \quad \{T.val := E.val\} \\
T &\rightarrow \text{num} \quad \{T.val := num.val\}
\end{aligned}$$

**Hình 5.21** - Lược đồ dịch đã được chuyển đổi có văn phạm đệ quy phải

Chẳng hạn đánh giá biểu thức  $9 - 5 + 2$



**Hình 5.22** - Xác định giá trị của biểu thức  $9-5+2$

**Ví dụ 5.16:** Xét lược đồ dịch xây dựng cây cú pháp cho biểu thức

$$\begin{aligned}
E &\rightarrow E_1 + T && \{E.nptr := mknod('+', E_1.nptr, T.nptr)\} \\
E &\rightarrow E_1 - T && \{E.nptr := mknod('-', E_1.nptr, T.nptr)\} \\
E &\rightarrow T && \{E.nptr := T.nptr\} \\
T &\rightarrow (E) && \{T.nptr := E.nptr\} \\
T &\rightarrow \text{id} && \{T.nptr := mkleaf(id, id.entry)\}
\end{aligned}$$

$$T \rightarrow \text{num} \quad \{T.nptr := mkleaf(num, num.val) \}$$

Áp dụng quy tắc khử đệ quy trái trên với  $E \approx A$ ,  $+T$ ,  $-T \approx Y$  và  $T \approx X$  ta có lược đồ dịch

$$\begin{aligned} E &\rightarrow T \quad \{R.i := T.nptr \} \\ &\quad R \quad \{E.nptr := R.s \} \\ R &\rightarrow + \\ &\quad T \quad \{R_l.i := mknnode('+', R.nptr, T.nptr) \} \\ &\quad R_l \quad \{R.s := R_l.s \} \\ R &\rightarrow - \\ &\quad T \quad \{R_l.i := mknnode('-', R.nptr, T.nptr) \} \\ &\quad R_l \quad \{R.s := R_l.s \} \\ R &\rightarrow \varepsilon \quad \{R.s := R.i \} \\ T &\rightarrow ( \\ &\quad E \\ &\quad ) \quad \{T.nptr := E.nptr \} \\ T &\rightarrow \text{id} \quad \{T.nptr := mkleaf(id, id.entry) \} \\ T &\rightarrow \text{num} \quad \{T.nptr := mkleaf(num, num.val) \} \end{aligned}$$

**Hình 5.23** - Lược đồ dịch được chuyển đổi để xây dựng cây cú pháp

## 2. Thiết kế bộ dịch dự đoán

**Giải thuật:** Xây dựng bộ dịch trực tiếp cú pháp dự đoán (Predictive - Syntax - Directed Translation)

**Input:** Một lược đồ dịch cú pháp trực tiếp với văn phạm có thể phân tích dự đoán.

**Output:** Mã cho bộ dịch trực tiếp cú pháp.

### Phương pháp:

1. Với mỗi ký hiệu chưa kết thúc  $A$ , xây dựng một hàm có các tham số hình thức tương ứng với các thuộc tính kế thừa của  $A$  và trả về giá trị của thuộc tính tổng hợp của  $A$ .
2. Mã cho ký hiệu chưa kết thúc  $A$  quyết định luật sinh nào được dùng cho ký hiệu nhập hiện hành.
3. Mã kết hợp với mỗi luật sinh như sau: chúng ta xem xét token, ký hiệu chưa kết thúc và hành vi bên phải của luật sinh từ trái sang phải
  - i) Đối với token  $X$  với thuộc tính tổng hợp  $x$ , lưu giá trị của  $x$  vào trong biến được khai báo cho  $X.x$ . Sau đó phát sinh lời gọi đệ hợp thức (match) token  $X$  và dịch chuyển đầu đọc.

ii) Đối với ký hiệu chưa kết thúc B, phát sinh lệnh gán  $C := B(b_1, b_2, \dots, b_k)$  với lời gọi hàm trong vế phải của lệnh gán, trong đó  $b_1, b_2, \dots, b_k$  là các biến cho các thuộc tính kế thừa của B và C là biến cho thuộc tính tổng hợp của B.

iii) Đối với một hành vi, chép mã vào trong bộ phân tích cú pháp, thay thế mỗi tham chiếu tới một thuộc tính bởi biến cho thuộc tính đó.

**Ví dụ 5.17:** Xét lược đồ dịch cho việc xây dựng cây cú pháp cho biểu thức. Ta thấy đó là một văn phạm LL(1) nên phù hợp cho phân tích trên xuống. Với 3 ký hiệu chưa kết thúc E, R, T ta xây dựng 3 hàm tương ứng:

```
function E: ↑ syntax - tree - node;      /* E không có thuộc tính kế thừa */
function R (i : ↑ syntax - tree - node) : ↑ syntax - tree - node
function T : ↑ syntax - tree - node;
```

Dùng token addop biểu diễn cho + và - ta có thể kết hợp hai luật sinh thành một luật sinh mới.

```
R → addop
    T      {  $R_1.i := mknode(addop.lexeme, R.i, T.nptr)$  }
    R1    {  $R.s := R_1.s$  }
R → ε      {  $R.s := R.i$  }
```

Ta có hàm R như sau:

```
function R(i : ↑ syntax_tree_node) : ↑ syntax_tree_node;
var nptr, i1, s1, s : ↑ syntax_tree_node;
    addoplexeme : char;
begin
    if lookahead = addop then
        begin /* luật sinh R → addop TR */
            addoplexeme := lexval;
            match(addop);
            nptr := T;
            i1 := mknode(addoplexeme, i, nptr);
            s1 := R(i1);
            s := s1;
        end
    else s := i; /* Luật sinh R → ε */
    return s;
end;
```

**Hình 5.24** - Xây dựng cây cú pháp đệ quy giảm

## BÀI TẬP CHƯƠNG V

5.1. Xây dựng một cây phân tích cú pháp chú thích cho biểu thức số học sau:

$$(4 * 7 + 1) * 2$$

5.2. Xây dựng một cây phân tích cú pháp và cây cú pháp cho biểu thức  $((a) + (b))$  theo:

- a) Định nghĩa trực tiếp cú pháp cho biểu thức số học.
- b) Lược đồ dịch cho biểu thức số học.

5.3. Xây dựng một DAG cho các biểu thức sau đây:

a)  $a + a + (a + a + a + (a + a + a + a))$

b)  $x * (3 * x + x * x)$

5.4. Văn phạm sau đây sinh ra các biểu thức có được khi áp dụng một toán tử số học + cho các hằng số nguyên và số thực. Khi 2 số nguyên được công lại, kiểu kết quả là kiểu nguyên, ngược lại nó là kiểu số thực.

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

$$T \rightarrow \text{num} \bullet \text{num} \mid \text{num}$$

- a) Đưa ra một định nghĩa trực tiếp cú pháp xác định kiểu của mỗi biểu thức con.
- b) Mở rộng định nghĩa trực tiếp cú pháp trên để dịch các biểu thức thành ký pháp hậu tố và xác định các kiểu. Dùng toán tử một ngôn inttoreal để chuyển một giá trị nguyên thành giá trị thực tương đương mà nhờ đó cả hai toán hạng của + ở dạng hậu tố có cùng kiểu.

5.5. Giả sử các khai báo sinh bởi văn phạm sau:

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

$$L \rightarrow , \text{id } L \mid : T$$

$$T \rightarrow \text{integer} \mid \text{real}$$

- a) Xây dựng một lược đồ dịch để nhập kiểu của mỗi định danh vào bảng danh biểu.
- b) Xây dựng chương trình dịch dự đoán từ lược đồ dịch trên.

**5.6.** Cho văn phạm sinh ra các dòng text như sau:

$$S \rightarrow L$$

$$L \rightarrow L B \mid B$$

$$B \rightarrow B \text{ sub } F \mid F$$

$$F \rightarrow \{ L \} \mid \text{text}$$

- a) Xây dựng một định nghĩa trực tiếp cú pháp cho văn phạm.
- b) Chuyển định nghĩa trực tiếp cú pháp trên thành lược đồ dịch.
- c) Loại bỏ đệ quy trái trong lược đồ dịch vừa xây dựng.



## CHƯƠNG VI

### KIỂM TRA KIỂU

#### Nội dung chính:

Hai cách kiểm tra kiểu là *kiểm tra tĩnh* được thực hiện trong thời gian biên dịch chương trình nguồn và *kiểm tra động* được thực hiện trong thời gian thực thi chương trình đích. Trong chương này ta tập trung vào phần xử lý ngữ nghĩa bằng cách kiểm tra tĩnh mà cụ thể là *kiểm tra kiểu*. Phần đầu của chương trình bày các khái niệm về *hệ thống kiểu*, *các biểu thức kiểu*. Phần còn lại mô tả cách tạo ra một *bộ kiểm tra kiểu* đơn giản.

#### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải nắm được:

- Hệ thống kiểu với các biểu thức kiểu (kiểu cơ sở và kiểu có cấu trúc) thường gặp ở bất cứ một ngôn ngữ lập trình nào.
- Dịch trực tiếp cú pháp cài đặt bộ kiểm tra kiểu đơn giản từ đó có thể mở rộng để cài đặt cho những ngôn ngữ phức tạp hơn.

#### Kiến thức cơ bản:

Sinh viên phải biết một số ngôn ngữ lập trình cấp cao như Pascal, C++, Java, v.v hoặc đã được học môn ngôn ngữ lập trình (phần đề cập đến các kiểu cơ sở và kiểu có cấu trúc).

#### Tài liệu tham khảo:

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

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

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

## I. HỆ THỐNG KIỂU

Trong các ngôn ngữ nói chung đều có kiểu cơ sở và kiểu có cấu trúc. Chẳng hạn trang Pascal, kiểu cơ sở là: boolean, char, integer, real, kiểu miền con và kiểu liệt kê. Các kiểu có cấu trúc như mảng, mẫu tin, tập hợp, ...

### 1. Biểu thức kiểu

Biểu thức kiểu bao gồm:

1. Kiểu cơ sở là một biểu thức kiểu: boolean, char, integer, real. Ngoài ra còn có các kiểu cơ sở đặc biệt như: kiểu `type_error`: chỉ ra một lỗi trong quá trình kiểm tra kiểu; kiểu `void`, “không có giá trị”, cho phép kiểm tra kiểu đối với lệnh.

2. Vì biểu thức kiểu có thể được đặt tên, tên kiểu là một biểu thức kiểu.
3. Cấu trúc kiểu là một biểu thức kiểu, các cấu trúc bao gồm:
  - a. **Mảng** (array): Nếu T là một biểu thức kiểu thì  $\text{array}(I, T)$  là một biểu thức kiểu. Một mảng có tập chỉ số I và các phần tử có kiểu T.
  - b. **Tích** (products): Nếu T1, T2 là biểu thức kiểu thì tích Decas  $T1 * T2$  là biểu thức kiểu.
  - c. **Mẫu tin** (records): Là cấu trúc bao gồm một bộ các tên trường, kiểu trường.
  - d. **Con trỏ** (pointers): Nếu T là một biểu thức kiểu thì  $\text{pointer}(T)$  là một biểu thức kiểu T.
  - e. **Hàm** (functions): Một cách toán học, hàm ánh xạ các phần tử của tập xác định (domain) lên tập giá trị (range). Một hàm là một biểu thức kiểu  $D \rightarrow R$

## 2. Hệ thống kiểu

Hệ thống kiểu là một bộ sưu tập các quy tắc để gán các biểu thức kiểu vào các phần của một chương trình. Bộ kiểm tra kiểu cài đặt một hệ thống kiểu.

## 3. Kiểm tra kiểu tĩnh và động

Kiểm tra được thực hiện bởi chương trình dịch được gọi là kiểm kiểu tĩnh. Kiểm tra được thực hiện trong khi chạy chương trình đích gọi là kiểm tra kiểu động.

## II. ĐẶC TẢ MỘT BỘ KIỂM TRA KIỂU ĐƠN GIẢN

Trong phần này chúng ta mô tả một bộ kiểm tra kiểu cho một ngôn ngữ đơn giản trong đó kiểu của mỗi một danh biểu phải được khai báo trước khi sử dụng. Bộ kiểm tra kiểu là một lược đồ dịch mà nó tổng hợp kiểu của mỗi biểu thức từ kiểu của các biểu thức con của nó.

### 1. Một ngôn ngữ đơn giản

Văn phạm sau sinh ra một chương trình, biểu diễn bởi một ký hiệu chưa kết thúc P chứa một chuỗi các khai báo D và một biểu thức đơn giản E.

$$P \rightarrow D ; E$$

$$D \rightarrow D ; D \mid \text{id} : T$$

$$T \rightarrow \text{char} \mid \text{integer} \mid \text{array}[\text{num}] \text{ of } T_1 \mid \uparrow T_1$$

$$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E_1 \text{ mod } E_2 \mid E_1 [E_2] \mid E_1 \uparrow$$

**Hình 6.1** - Văn phạm của một ngôn ngữ đơn giản

- Các kiểu cơ sở: char, integer và type-error
- Mảng bắt đầu từ 1. Chẳng hạn  $\text{array}[256] \text{ of char}$  là biểu thức kiểu (1...256, char)
- Kiểu con trỏ  $\uparrow T$  là một biểu thức kiểu  $\text{pointer}(T)$ .

Ta có lược đồ dịch để lưu trữ kiểu của một danh biểu

$$P \rightarrow D ; E$$

$D \rightarrow D ; D$	
$D \rightarrow \text{id} : T$	$\{ \text{addtype}(\text{id.entry}, T.\text{type}) \}$
$T \rightarrow \text{char}$	$\{ T.\text{type} := \text{char} \}$
$T \rightarrow \text{integer}$	$\{ T.\text{type} := \text{integer} \}$
$T \rightarrow \uparrow T_1$	$\{ T.\text{type} := \text{pointer}(T_1.\text{type}) \}$
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$	$\{ T.\text{type} := \text{array}(1 \dots \text{num.val}, T_1.\text{type}) \}$

**Hình 6.2-** *Lược đồ dịch lưu trữ kiểu của một danh biểu*

## 2. Kiểm tra kiểu của biểu thức

Lược đồ dịch cho kiểm tra kiểu của biểu thức như sau:

$E \rightarrow \text{literal}$	$\{ E.\text{type} := \text{char} \}$
$E \rightarrow \text{num}$	$\{ E.\text{type} := \text{integer} \}$
$E \rightarrow \text{id}$	$\{ E.\text{type} := \text{lookup}(\text{id.entry}) \}$
$E \rightarrow E_1 \text{ mod } E_2$	$\{ E.\text{type} := \text{if } E_1.\text{type} = \text{integer and } E_2.\text{type} = \text{integer} \\ \text{then integer else type\_error} \}$
$E \rightarrow E_1[E_2]$	$\{ E.\text{type} := \text{if } E_2.\text{type} = \text{integer and } E_1.\text{type} = \text{array}(s,t) \\ \text{then } t \text{ else type\_error} \}$
$E \rightarrow E_1 \uparrow$	$\{ E.\text{type} := \text{if } E_1.\text{type} = \text{pointer}(t) \text{ then } t \\ \text{else type\_error} \}$

**Hình 6.3-** *Lược đồ dịch kiểm tra kiểu của biểu thức*

Ở đây ta dùng hàm  $\text{lookup}(e)$  để tìm kiểu được lưu trữ trong ô của bảng ký hiệu mà ô đó được trỏ bởi  $e$ .

## 3. Kiểm tra kiểu của các lệnh

Ta có lược đồ dịch cho kiểm tra kiểu của lệnh

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

**Hình 6.4-** *Lược đồ dịch kiểm tra kiểu của các lệnh*

#### 4. Kiểm tra kiểu của các hàm

Áp dụng hàm vào một đối số có thể được cho bởi luật sinh  $E \rightarrow E(E)$ . Lược đồ dịch cho kiểm tra kiểu cho một áp dụng hàm là:

$$E \rightarrow E_1(E_2) \quad \{E.type := \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \\ \text{else type\_error} \}$$

**Hình 6.5-** Lược đồ dịch kiểm tra kiểu của hàm

Luật sinh trên biểu diễn rằng một biểu thức được hình thành áp dụng  $E_1$  lên  $E_2$ , kiểu của  $E_1$  phải là một hàm  $s \rightarrow t$  từ kiểu  $s$  của  $E_2$  tới một kiểu giới hạn  $t$  nào đó; kiểu của  $E_1(E_2)$  là  $t$ .

### III. SỰ TƯƠNG ĐƯƠNG CỦA CÁC BIỂU THỨC KIỂU

Thông thường kiểm tra kiểu có dạng: “nếu hai biểu thức kiểu bằng nhau thì trả về một kiểu ngược lại trả về `type_error`”. Điều quan trọng là cần xác định khi nào hai biểu thức kiểu tương đương.

#### 1. Tương đương cấu trúc

Hai biểu thức kiểu được gọi là tương đương cấu trúc nếu cấu trúc của chúng giống hệt nhau.

**Ví dụ 6.1:**

- Biểu thức kiểu integer tương đương với integer vì chúng là một kiểu cơ sở.
- `pointer(integer)` tương đương với `pointer(integer)` vì cả hai được hình thành bằng cách áp dụng cùng một cấu trúc con trỏ pointer lên các kiểu tương đương.

Giả sử,  $s$  và  $t$  là hai biểu thức kiểu, hàm sau kiểm tra xem chúng có tương đương hay không?

**Function** *sequiv*( $s, t$ ) : **boolean**;

**Begin**

*if*  $s$  và  $t$  cùng là một kiểu cơ sở **then**

**return true**

**else if**  $s = \text{array}(s_1, s_2)$  **and**  $t = \text{array}(t_1, t_2)$  **then**

**return** *sequiv*( $s_1, t_1$ ) **and** *sequiv*( $s_2, t_2$ )

**else if**  $s = \text{pointer}(s_1)$  **and**  $t = \text{pointer}(t_1)$  **then**

**return** *sequiv*( $s_1, t_1$ )

**else if**  $s = s_1 \rightarrow s_2$  **and**  $t = t_1 \rightarrow t_2$  **then**

**return** *sequiv*( $s_1, t_1$ ) **and** *sequiv*( $s_2, t_2$ )

**else return false;**

**end;**

**Hình 6.6-** Đoạn ngôn ngữ giả kiểm tra sự tương đương cấu trúc của hai biểu thức kiểu  $s$  và  $t$

## 2. Tương đương tên

Trong một số ngôn ngữ, kiểu được cho bởi tên. Ví dụ trong Pascal

```
type link = ↑ cell;
var next : link;
    last : link;
    p : ↑ cell;
    q, r : ↑ cell;
```

Danh biểu link được khai báo là tên của kiểu  $\uparrow \text{cell}$ . Vấn đề đặt ra là next, last, p, q, r có kiểu giống nhau hay không? Câu trả lời phụ thuộc vào sự cài đặt. Hai biểu thức kiểu là tương đương tên nếu tên của chúng giống nhau. Theo quan niệm tương đương tên thì last và next có cùng kiểu; p, q và r có cùng một kiểu nhưng next và p có kiểu khác nhau.

## IV. CHUYỂN ĐỔI KIỂU

Xét biểu thức  $x + i$  trong đó  $x$  có kiểu real và  $i$  có kiểu integer. Vì biểu diễn các số nguyên, số thực khác nhau trong máy tính do đó các chỉ thị máy khác nhau được dùng cho số thực và số nguyên. Trình biên dịch có thể thực hiện việc chuyển đổi kiểu để hai toán hạng có cùng kiểu khi phép toán cộng xảy ra.

Bộ kiểm tra kiểu trong trình biên dịch có thể được dùng để thêm các phép toán biến đổi kiểu vào trong biểu diễn trung gian của chương trình nguồn. Chẳng hạn ký hiệu hậu tố của  $x + i$  có thể là:  **$x \ i \ \text{inttoreal} \ \text{real+}$**

Trong đó: **inttoreal** đổi số nguyên  $i$  thành số thực, **real+** thực hiện phép cộng các số thực.

*Sự ép buộc chuyển đổi kiểu*

Sự chuyển đổi từ kiểu này sang kiểu khác được gọi là ẩn (implicit) nếu nó được làm một cách tự động bởi chương trình dịch. Chuyển đổi kiểu ẩn còn gọi là ép buộc chuyển đổi kiểu (coercions).

**Ví dụ 6.2:** Định nghĩa trực tiếp cú pháp cho kiểm tra kiểu và ép buộc chuyển đổi kiểu biến đổi kiểu từ integer thành real:

Luật sinh	Luật ngữ nghĩa
$E \rightarrow \text{num}$	$E.type := integer$
$E \rightarrow \text{num.num}$	$E.type := real$
$E \rightarrow \text{id}$	$E.type := lookup(id.entry)$
$E \rightarrow E_1 \text{ op } E_2$	$E.type := \text{if } E_1.type = integer \text{ and } E_2.type = integer$ $\quad \quad \quad \text{then } integer$ $\quad \quad \quad \text{else if } E_1.type = integer \text{ and } E_2.type = real$

```

        then real
    else if E1.type = real and E2.type = integer
        then real
    else if E1.type = real and E2.type = real
        then real
    else type_error

```

**Hình 6.7-** Định nghĩa trực tiếp cú pháp cho kiểm tra kiểu và ép buộc chuyển đổi kiểu

Chú ý rằng việc ép buộc chuyển đổi kiểu có thể dẫn đến sự lãng phí thời gian thực hiện chương trình.

**Ví dụ 6.3:** Với khai báo x là một mảng các số thực thì lệnh for i:=1 to n do x[i]:=1 thực hiện trong 48,4 micro giây còn lệnh for i:=1 to n do x[i]:=1.0 thực hiện trong 5,4 micro giây. Sở dĩ như vậy vì mã phát sinh cho lệnh thứ nhất chứa một lời gọi thủ tục đổi số nguyên thành số thực tại thời gian thực hiện.

## BÀI TẬP CHƯƠNG VI

**6.1.** Viết các biểu thức kiểu cho các kiểu dữ liệu sau đây:

a) Một mảng của các con trỏ có kích thước từ 1 đến 100, trỏ đến đối tượng các số thực.

b) Mảng 2 chiều của các số nguyên, hàng có kích thước từ 0 đến 9, cột có chỉ số từ -10 đến 10.

c) Các hàm mà miền định nghĩa là các hàm với các đối số nguyên, trị là con trỏ trỏ đến các số nguyên và miền xác định của nó là các mẫu tin chứa số nguyên và ký tự.

**6.2.** Giả sử có một khai báo C như sau:

```
typedef struct {
    int a, b ;
} CELL, * PCELL ;
CELL    foo [ 100 ] ;
PCELL    bar (x, y)    int x ; CELL y { ..... }
```

Viết các biểu thức kiểu cho các kiểu dữ liệu foo và bar.

**6.3.** Cho văn phạm sau đây định nghĩa chuỗi của các chuỗi ký tự:

```
P → D ; E
D → D ; D | id : T
T → list of T | char | integer
E → ( L ) | literal | num | id
L → E , L | E
```

Hãy viết các quy tắc biên dịch để xác định các biểu thức kiểu (E) và list (L).

**6.4.** Giả sử tên kiểu là link và cell được định nghĩa như ở phần tên cho biểu thức kiểu. Hãy xác định những biểu thức kiểu nào dưới đây là tương đương cấu trúc, những biểu thức kiểu nào tương đương tên.

a) **link**

b) *pointer* (**cell**)

c) *pointer* (**link**)

d) *pointer* (*record* ((*info* x *integer*) x ( *next* x *pointer* (**cell**))))

**6.5.** Giả sử rằng kiểu của mỗi định danh là một miền con của số nguyên. Cho biểu thức với các phép toán +, -, \*, div và mod như trong Pascal, hãy viết quy tắc kiểm tra kiểu để gán mỗi biểu thức con vào vùng miền con giá trị mà nó sẽ nằm trong đó.

## CHƯƠNG VII

### MÔI TRƯỜNG THỜI GIAN THỰC HIỆN

#### **Nội dung chính:**

Trước khi xem xét vấn đề sinh mã được trình bày ở các chương sau, chương này trình bày một số vấn đề liên quan đến việc *gọi thực hiện chương trình con*, các *chiến lược cấp phát bộ nhớ* và *quản lý bảng ký hiệu*. Cùng một tên trong chương trình nguồn có thể biểu thị cho nhiều *đối tượng dữ liệu* trong chương trình đích. Sự biểu diễn của các đối tượng dữ liệu tại thời gian thực thi được xác định bởi kiểu của nó. Sự cấp phát và thu hồi các đối tượng dữ liệu được quản lý bởi một tập các *chương trình con ở dạng mã đích*. Việc thiết kế các chương trình con này được xác định bởi ngữ nghĩa của chương trình nguồn. Mỗi sự thực thi của một chương trình con được gọi là một *mẫu tin kích hoạt*. Nếu một chương trình con đệ quy, một số mẫu tin kích hoạt có thể tồn tại cùng một thời điểm. Mỗi ngôn ngữ lập trình đều có *quy tắc tầm vực* để xác định việc xử lý khi tham khảo đến các tên không cục bộ. Tùy vào ngôn ngữ, nó cho phép một *chương trình chứa các chương trình con lồng nhau hoặc không lồng nhau*; Cho phép *gọi đệ quy hoặc không đệ quy*; Cho phép *truyền tham số* bằng giá trị hay tham chiếu ... Vì thế, khi thiết kế một chương trình ở dạng mã đích ta cần chú ý đến các yếu tố này.

#### **Mục tiêu cần đạt:**

Sau khi học xong chương này, sinh viên phải nắm được:

- Cách gọi và thực thi một chương trình.
- Cách tổ chức bộ nhớ và các chiến lược cấp phát – thu hồi bộ nhớ.

#### **Kiến thức cơ bản:**

Sinh viên phải biết một số ngôn ngữ lập trình cấp cao như Pascal, C++, Java, v.v hoặc đã được học môn ngôn ngữ lập trình (phần đề cập đến các chương trình con).

#### **Tài liệu tham khảo:**

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

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

## **I. CHƯƠNG TRÌNH CON**

### **1. Định nghĩa chương trình con**

Định nghĩa chương trình con là một sự khai báo nó. Dạng đơn giản nhất là sự kết hợp giữa tên chương trình con và thân của nó.

**Ví dụ 7.1:** Chương trình Pascal đọc và sắp xếp các số nguyên



```

(1)  program sort(input, output)
(2)      var a: array[0..10] of integer;
(3)  procedure readarray;
(4)      var i: integer;
(5)      begin
(6)          for i=1 to 9 do read(a[i]);
(7)      end;
(8)  function partition(y,z:integer): integer;
(9)      var i,j,x,v: integer;
(10)     begin...
(11)     end;
(12) procedure quicksort(m,n:integer);
(13)     var i: integer;
(14)     begin;
(15)         if (n>m) then begin
(16)             i:= partition(m,n);
(17)             quicksort(m,i-1);
(18)             quicksort(i+1,n);
(19)         end;
(20)     end;
(21)     begin
(22)         a[0]:= -9999, a[10]:= 9999;
(23)         readarray;
(24)         quicksort(1,9);
(25)     end.

```

**Hình 7.1-** Chương trình Pascal đọc và sắp xếp các số nguyên

Chương trình trên chứa các định nghĩa chương trình con

- Chương trình con readarray từ dòng 3 - 7, thân của nó từ 5 - 7
- Chương trình con partition từ dòng 8 - 11, thân của nó từ 10 - 11.
- Chương trình con quicksort từ dòng 12 - 20, thân của nó từ 14 - 20.

Chương trình chính cũng được xem như là một chương trình con có thân từ dòng 21 - 25

Khi tên chương trình con xuất hiện trong phần thân của một chương trình con ta nói chương trình con được gọi tại điểm đó.

## 2. Cây hoạt động

Trong quá trình thực hiện chương trình thì:

1. Dòng điều khiển là tuần tự: tức là việc thực hiện chương trình bao gồm một chuỗi các bước. Tại mỗi bước đều có một sự điều khiển xác định.

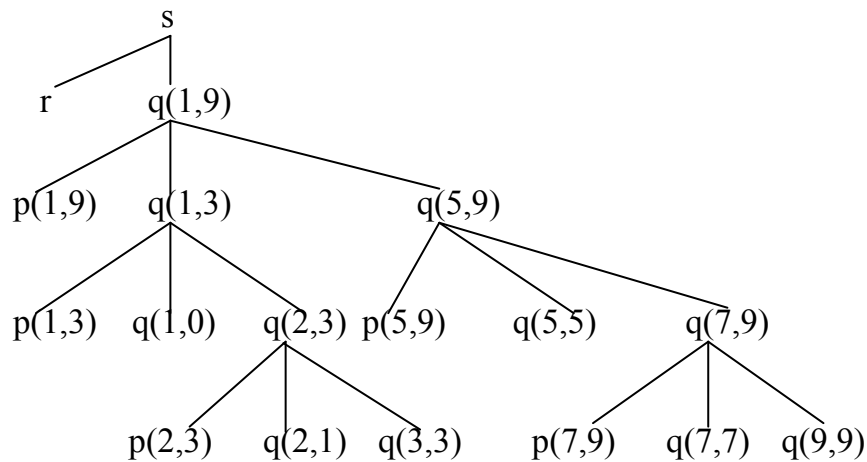
2. Việc thực hiện chương trình con bắt đầu tại điểm bắt đầu của thân chương trình con và trả điều khiển về cho chương trình gọi tại điểm nằm sau lời gọi khi việc thực hiện chương trình con kết thúc.

- Thời gian tồn tại của một chương trình con  $p$  là một chuỗi các bước giữa bước đầu tiên và bước cuối cùng trong sự thực hiện thân chương trình con bao gồm cả thời gian thực hiện các chương trình con được gọi bởi  $p$ .
- Nếu  $a$  và  $b$  là hai sự hoạt động của hai chương trình con tương ứng thì thời gian tồn tại của chúng tách biệt nhau hoặc lồng nhau.
- Một chương trình con là đệ quy nếu một hoạt động mới có thể bắt đầu trước khi một hoạt động trước đó của chương trình con đó kết thúc.
- Để đặc tả cách thức điều khiển vào ra mỗi hoạt động của chương trình con ta dùng cấu trúc cây gọi là cây hoạt động.
  1. Mỗi nút biểu diễn cho một hoạt động của một chương trình con.
  2. Nút gốc biểu diễn cho hoạt động của chương trình chính.
  3. Nút  $a$  là cha của  $b$  nếu và chỉ nếu dòng điều khiển sự hoạt động đó từ  $a$  sang  $b$
  4. Nút  $a$  ở bên trái của nút  $b$  nếu thời gian tồn tại của  $a$  xuất hiện trước thời gian tồn tại của  $b$ .

**Ví dụ 7.2:** Xét chương trình sort nói trên

- Bắt đầu thực hiện chương trình.
- Vào readarray.
- Ra khỏi readarray.
- Vào quicksort(1,9).
- Vào partition(1,9)
- Ra khỏi partition(1,9) // giả sử trả về 4
- Vào quicksort(1,3)
- . . . . .
- Ra khỏi quicksort(1,3).
- Vào quicksort(5,9);
- . . . . .
- Ra khỏi quicksort(5,9).
- Sự thực hiện kết thúc.

**Hình 7.2** - Xuất các mẫu tin hoạt động đề nghị của chương trình trong hình 7.1  
Ta có cây hoạt động tương ứng.



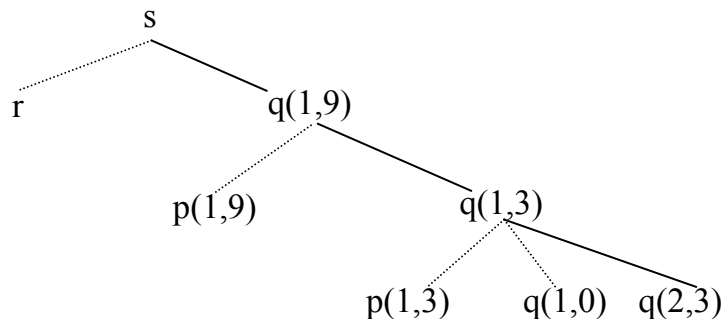
**Hình 7.3**- Cây hoạt động tương ứng với phần xuất trong hình 7.2

### 3. Ngăn xếp điều khiển

Dòng điều khiển một chương trình tương ứng với phép duyệt theo chiều sâu của cây hoạt động. Bắt đầu từ nút gốc, thăm một nút trước các con của nó và thăm các con một cách đệ quy tại mỗi nút từ trái sang phải.

Chúng ta có thể dùng một Stack, gọi là Stack điều khiển, để lưu trữ sự hoạt động của chương trình con. Khi sự hoạt động của một chương trình con bắt đầu thì đẩy nút tương ứng với sự hoạt động đó lên đỉnh Stack. Khi sự hoạt động kết thúc thì pop nút đó ra khỏi Stack. Nội dung của Stack thể hiện đường dẫn đến nút gốc của cây hoạt động. Khi nút  $n$  nằm trên đỉnh Stack thì Stack chứa các nút nằm trên đường từ  $n$  đến gốc.

**Ví dụ 7.3:** Hình sau trình bày nội dung của Stack đang lưu trữ đường đi từ nút  $q(2, 3)$  đến nút gốc. Các cạnh nét đứt thể hiện một nút đã pop ra khỏi Stack.



**Hình 7.4** - Stack điều khiển chứa các nút dẫn tới nút gốc

Tại thời điểm này thì đường dẫn tới gốc là:  $s \ q(1, 9) \ q(1, 3) \ q(2, 3)$  ( $q(2, 3)$  nằm trên đỉnh Stack)

#### 4. Tầm vực của sự khai báo

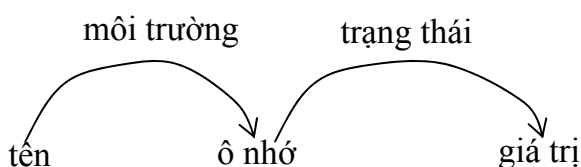
Đoạn chương trình chịu ảnh hưởng của một sự khai báo gọi là tầm vực của khai báo đó.

Trong một chương trình có thể có nhiều sự khai báo trùng tên ví dụ biến  $i$  trong chương trình sort. Các khai báo này độc lập với nhau và chịu sự chi phối bởi quy tắc tầm của ngôn ngữ.

Sự xuất hiện của một tên trong một chương trình con được gọi là cục bộ (local) trong chương trình con ấy nếu tầm vực của sự khai báo nằm trong chương trình con, ngược lại được gọi là không cục bộ (nonlocal).

#### 5. Liên kết tên

Trong ngôn ngữ của ngôn ngữ lập trình, thuật ngữ môi trường (environment) để chỉ một ánh xạ từ một tên đến một vị trí ô nhớ và thuật ngữ trạng thái (state) để chỉ một ánh xạ từ vị trí ô nhớ tới giá trị lưu trữ trong đó



**Hình 7.5** - Hai ánh xạ từ tên tới giá trị

Môi trường khác trạng thái: một lệnh gán làm thay đổi trạng thái nhưng không thay đổi môi trường.

Khi một môi trường kết hợp vị trí ô nhớ  $s$  với một tên  $x$  ta nói rằng  $x$  được liên kết tới  $s$ . Sự kết hợp đó được gọi là mối liên kết của  $x$ .

Liên kết là một bản sao động (dynamic counterpart) của sự khai báo.

Chúng ta có sự tương ứng giữa các ký hiệu động và tĩnh:

Ký hiệu tĩnh	Bản sao động
Định nghĩa chương trình con	Sự hoạt động của chương trình con
Sự khai báo tên	Liên kết của tên
Tầm vực của sự khai báo	Thời gian tồn tại của liên kết

**Hình 7.6** - Sự tương ứng giữa ký hiệu động và tĩnh

#### 6. Các vấn đề cần quan tâm khi làm chương trình dịch

Các vấn đề cần đặt ra khi tổ chức lưu trữ và liên kết tên:

1. Chương trình con có thể đệ quy không?
2. Điều gì xảy ra cho giá trị của các tên cục bộ khi trả điều khiển từ hoạt động của một chương trình con.

3. Một chương trình con có thể tham khảo tới một tên cục bộ không?
4. Các tham số được truyền như thế nào khi gọi chương trình con.
5. Một chương trình con có thể được truyền như một tham số?
6. Một chương trình con có thể được trả về như một kết quả?
7. Bộ nhớ có được cấp phát động không?
8. Bộ nhớ có phải giải phóng một cách tường minh?

## II. TỔ CHỨC BỘ NHỚ

Tổ chức bộ nhớ trong thời gian thực hiện ở đây có thể sử dụng cho các ngôn ngữ Fortran, Pascal và C.

### 1. Phân chia bộ nhớ trong thời gian thực hiện

Bộ nhớ có thể chia ra để lưu trữ các phần:

1. Mã đích.
2. Đối tượng dữ liệu.
3. Bản sao của Stack điều khiển để lưu trữ hoạt động của chương trình con.

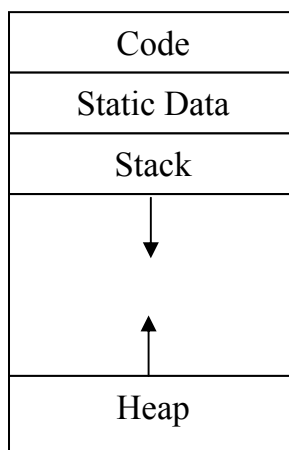
Trong đó kích thước của mã đích được xác định tại thời gian dịch cho nên nó được cấp phát tĩnh tại vùng thấp của bộ nhớ. Tương tự kích thước của một số đối tượng dữ liệu cũng có thể biết tại thời gian dịch cho nên nó cũng được cấp phát tĩnh.

Cài đặt các ngôn ngữ như Pascal, C dùng sự mở rộng của Stack điều khiển để quản lý sự hoạt động của chương trình con.

Khi có một lời gọi chương trình con, sự thể hiện của một hoạt động bị ngắt và thông tin về tình trạng của máy, chẳng hạn như giá trị bộ đếm chương trình (program counter) và thanh ghi được lưu vào trong Stack. Khi điều khiển trả về từ lời gọi, hoạt động này được tiếp tục sau khi khôi phục lại giá trị của thanh ghi và đặt bộ đếm chương trình vào ngay sau lời gọi.

Đối tượng dữ liệu mà thời gian tồn tại của nó được chứa trong một hoạt động được lưu trong Stack.

Một vùng khác của bộ nhớ được gọi là Heap lưu trữ tất cả các thông tin khác.



**Hình 7.7 - Phân chia bộ nhớ trong thời gian thực hiện cho mã đích và các vùng dữ liệu khác**

## 2. Mẫu tin hoạt động

Thông tin cần thiết để thực hiện một chương trình con được quản lý bằng cách dùng một mẫu tin hoạt động bao gồm một số trường như sau :

Giá trị trả về
Các tham số thực tế
Liên kết điều khiển
Liên kết truy nhập
Trạng thái máy
Dữ liệu cục bộ
Giá trị tạm thời

**Hình 7.8 - Mẫu tin hoạt động tổng quát**

Ý nghĩa các trường như sau:

1. Giá trị tạm thời: được lưu giữ trong quá trình đánh giá biểu thức.
2. Dữ liệu cục bộ: Lưu trữ dữ liệu cục bộ trong khi thực hiện chương trình con.
3. Trạng thái máy: lưu giữ thông tin về trạng thái của máy trước khi một chương trình con được gọi. Thông tin máy bao gồm bộ đếm chương trình và thanh ghi lệnh mà nó sẽ phục hồi khi điều khiển trả về từ chương trình con
4. Liên kết truy nhập: tham khảo tới dữ liệu không cục bộ được lưu trong các mẫu tin hoạt động khác.
5. Liên kết điều khiển: trở tới mẫu tin hoạt động của chương trình gọi.
6. Các tham số thực tế: được sử dụng bởi các chương trình gọi để cho chương trình được gọi. Thông thường các tham số được lưu trong thanh ghi chứ không phải trong mẫu tin hoạt động.
7. Giá trị trả về: được dùng bởi chương trình được gọi để trả về cho chương trình gọi một giá trị. Trong thực tế giá trị này thường được trả về trong thanh ghi.

## III. CHIẾN LƯỢC CẤP PHÁT BỘ NHỚ

Đối với các vùng nhớ khác nhau trong tổ chức bộ nhớ, ta có các chiến lược cấp phát khác nhau :

1. Cấp phát tĩnh cho tất cả các đối tượng dữ liệu tại thời gian dịch.

2. Cấp phát sử dụng Stack cho bộ nhớ trong thời gian thực hiện.
3. Đối với vùng dữ liệu Heap sử dụng cấp phát và thu hồi dạng Heap.

## 1. Cấp phát tĩnh


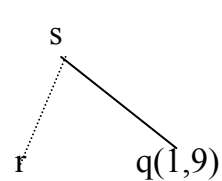
Trong cấp phát tĩnh, tên được liên kết với vùng nhớ lúc chương trình được dịch. Vì mỗi liên kết không thay đổi tại thời gian chạy nên mọi lần một chương trình con được kích hoạt, tên của nó được liên kết với cùng một vùng nhớ. Tính chất này cho phép giá trị của các tên cục bộ được giữ lại thông qua hoạt động của các chương trình con. Từ kiểu của tên, trình biên dịch xác định kích thước bộ nhớ của nó. Do đó trình biên dịch xác định được vị trí của mẫu tin kích hoạt giữa đoạn mã chương trình và các mẫu tin kích hoạt khác. Trong thời gian biên dịch, chúng ta có thể điền vào đoạn của các địa chỉ mà mã lệnh có thể tìm đến để truy xuất dữ liệu. Tương tự địa chỉ các vùng lưu trữ thông tin khi chương trình con được gọi đều được xác định tại thời gian dịch. Tuy nhiên cấp phát tĩnh cũng có một số hạn chế sau:

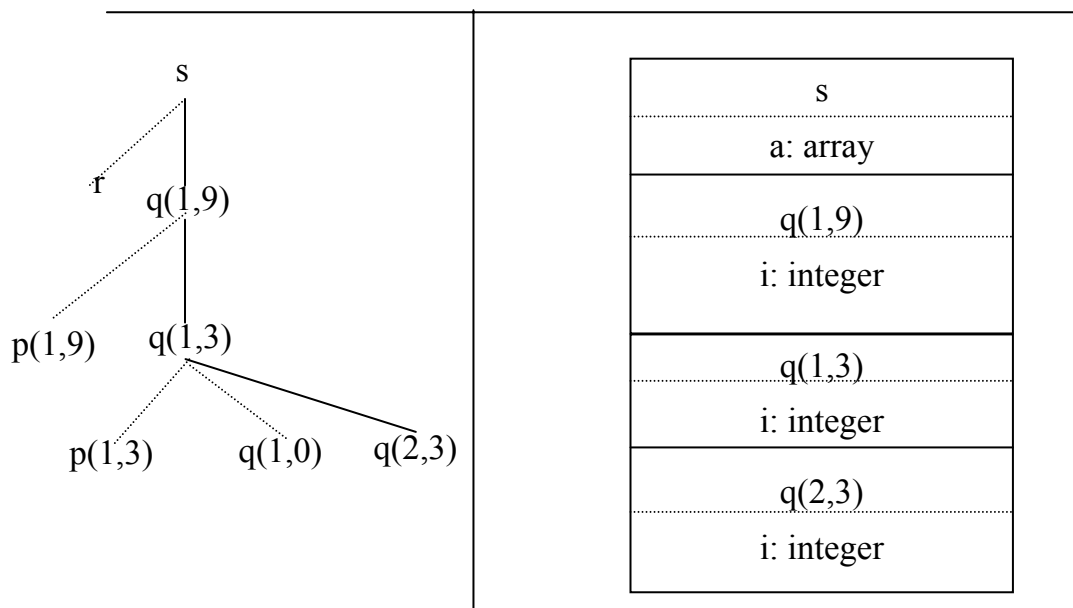
1. Kích thước và vị trí của đối tượng dữ liệu trong bộ nhớ phải được xác định tại thời gian dịch.
2. Không cho phép gọi đệ quy vì tất cả các kích hoạt của một chương trình con đều dùng chung một liên kết đối với tên cục bộ.
3. Cấu trúc dữ liệu không thể được cấp phát động vì không có cơ chế để cấp phát tại thời gian thực hiện.

## 2. Cấp phát ô nhớ sử dụng Stack

Bộ nhớ được tổ chức như là một Stack. Các mẫu tin kích hoạt được push vào Stack khi hoạt động bắt đầu và pop ra khỏi Stack khi hoạt động kết thúc.

**Ví dụ 7.4:** Chúng ta sẽ minh họa việc cấp phát và loại bỏ mẫu tin kích hoạt tương ứng với cây hoạt động của chương trình sort:

Cây hoạt động	Mẫu tin kích hoạt trong Stack
s	<div> <div>s</div> <div>a: array</div> </div>
	<div> <div>s</div> <div>a: array</div> <div>r</div> <div>i: integer</div> </div>
	<div> <div>s</div> <div>a: array</div> <div>q(1,9)</div> <div>i: integer</div> </div>



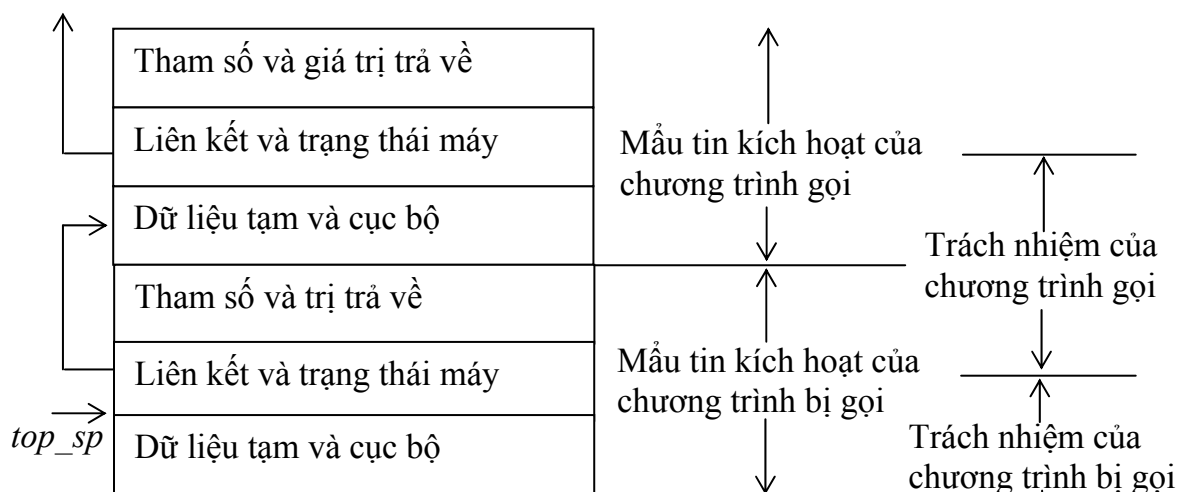
**Hình 7.9 - Sự cấp phát và loại bỏ các mẫu tin kích hoạt**

Bộ nhớ cho dữ liệu cục bộ trong mỗi lần gọi một chương trình con được chứa trong mẫu tin kích hoạt cho lần gọi đó. Như vậy các tên cục bộ được liên kết với bộ nhớ trong mỗi một hoạt động, bởi vì một mẫu tin kích hoạt được push vào Stack khi có một lời gọi chương trình con. Dữ liệu của các biến cục bộ sẽ bị xóa bỏ khi sự thực hiện chương trình con kết thúc.

Giả sử thanh ghi top đánh dấu đỉnh của Stack. Tại thời gian thực hiện một mẫu tin kích hoạt có thể được cấp phát hoặc thu hồi bằng cách tăng hoặc giảm thanh ghi top bằng kích thước của mẫu tin kích hoạt.

### Gọi thực hiện chương trình con

Gọi chương trình con được thực hiện bởi lệnh gọi trong mã đích - lệnh gọi cấp phát một mẫu tin kích hoạt và đưa thông tin vào cho các trường - lệnh trở về sẽ phục hồi các trạng thái máy để chương trình gọi tiếp tục thực hiện



**Hình 7.10 - Phân chia công việc giữa chương trình gọi và chương trình bị gọi**



Hình trên mô tả mối quan hệ giữa mẫu tin kích hoạt của chương trình gọi và chương trình bị gọi. Mỗi mẫu tin như vậy có ba trường chủ yếu: các tham số thực tế và trị trả về, các mối liên kết và trạng thái máy và cuối cùng là trường dữ liệu tạm và cục bộ.

Thanh ghi `top.sp` chỉ đến cuối trường các mối liên kết và trạng thái máy. Vị trí này được biết bởi chương trình gọi. Đoạn mã cho chương trình bị gọi có thể truy xuất dữ liệu tạm và cục bộ của nó bằng cách sử dụng độ dời (offsets) từ `top.sp`.

***Lệnh gọi thực hiện các công việc sau :***

1. Chương trình gọi đánh giá các tham số thực tế.
2. Chương trình gọi lưu địa chỉ trả về và giá trị cũ của `top.sp` vào trong mẫu tin kích hoạt của chương trình bị gọi. Sau đó tăng giá trị của `top.sp`.
3. Chương trình được gọi lưu giá trị thanh ghi và các thông tin trạng thái khác
4. Chương trình được gọi khởi tạo dữ liệu cục bộ của nó và bắt đầu thực hiện.

***Lệnh trả về thực hiện các công việc sau:***

1. Chương trình bị gọi gửi giá trị trả về vào mẫu tin kích hoạt của chương trình gọi.
2. Căn cứ vào thông tin trong trường trạng thái, chương trình bị gọi khôi phục `top_sp` cũng như giá trị các thanh ghi và truyền tới địa chỉ trả về trong mã của chương trình gọi.
3. Mặc dù `top_sp` đã bị giảm, chương trình gọi cần sao chép giá trị trả về vào trong mẫu tin kích hoạt của nó để sử dụng cho việc tính toán biểu thức.

**Dữ liệu có kích thước thay đổi**

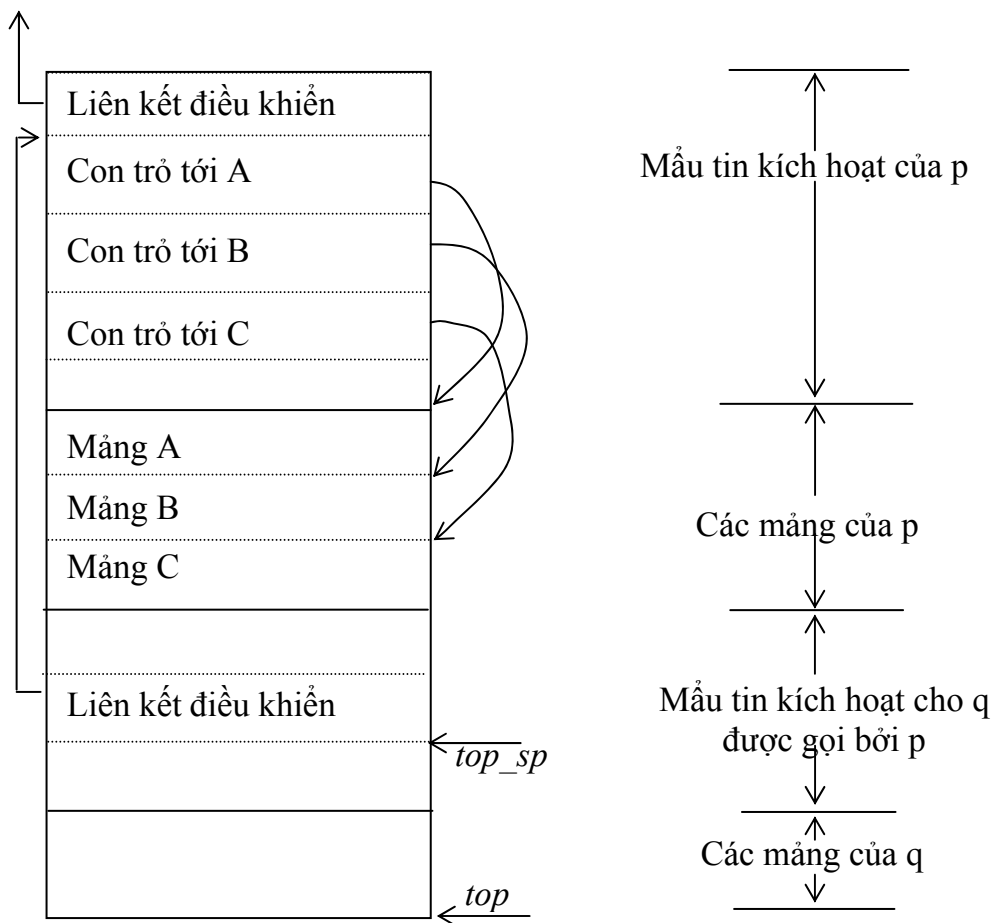
Một số ngôn ngữ cho phép dữ liệu có kích thước thay đổi.

Chẳng hạn chương trình con `p` có 3 mảng có kích thước thay đổi, các mảng này được lưu trữ ngoài mẫu tin kích hoạt của `p`. Trong mẫu tin kích hoạt của `p` chỉ chứa các con trỏ tới điểm bắt đầu của mỗi một mảng. Địa chỉ tương đối của các con trỏ này được biết tại thời gian dịch nên mã đích có thể truy nhập tới các phần tử mảng thông qua con trỏ.

Hình sau trình bày chương trình con `q` được gọi bởi `p`. Mẫu tin kích hoạt của `q` nằm sau các mảng của `p`. Truy nhập vào dữ liệu trong Stack thông qua hai con trỏ **`top`**, **`top.sp`**:

**`top`** chỉ đỉnh Stack nơi một mẫu tin kích hoạt mới có thể bắt đầu.

**`top_sp`** dùng để tìm dữ liệu cục bộ



**Hình 7.11 - Truy xuất các mảng được cấp phát động**

### 3. Cấp phát Heap

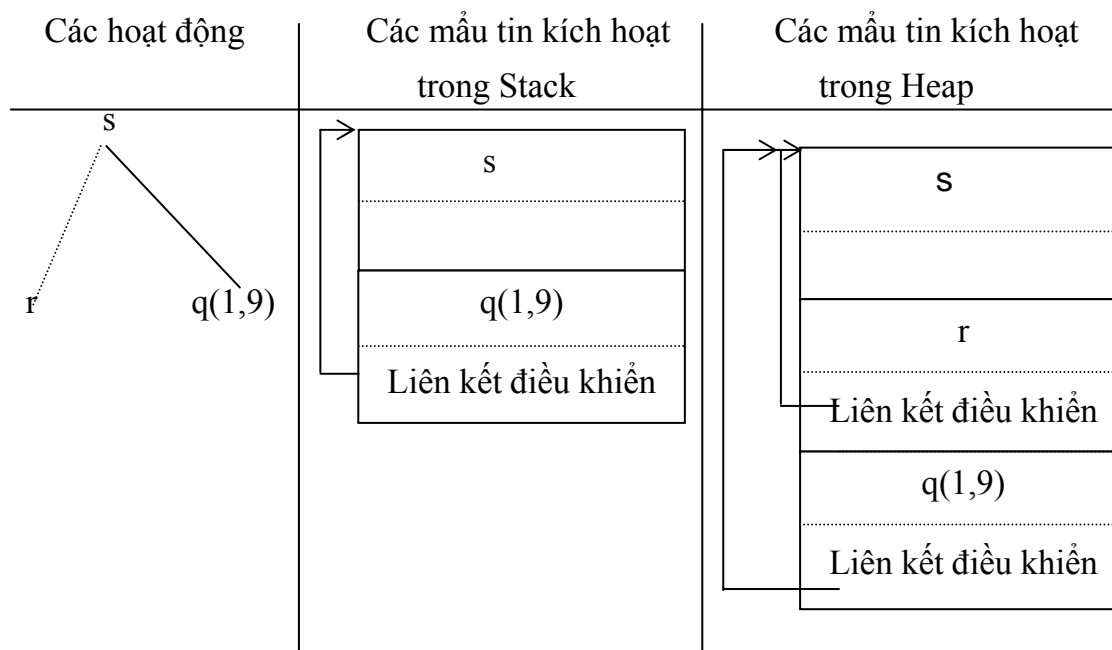
Chiến thuật cấp phát sử dụng Stack không đáp ứng được các yêu cầu sau:

1. Giá trị của tên cục bộ được giữ lại khi hoạt động của chương trình con kết thúc.
2. Hoạt động của chương trình bị gọi tồn tại sau chương trình gọi.

Các yêu cầu trên đều không thể cấp phát và thu hồi theo cơ chế LIFO (Last - In, First - Out) tức là tổ chức theo Stack.

Heap là khối ô nhớ liên tục được chia nhỏ để có thể cấp phát cho các mẫu tin kích hoạt hoặc các đối tượng dữ liệu khác.

Sự khác nhau giữa cấp phát Stack và Heap là ở chỗ mẫu tin cho một hoạt động được giữ lại khi hoạt động đó kết thúc.



**Hình 7.12** - Mẫu tin kích hoạt được giữ lại trong Heap

Về mặt vật lý, mẫu tin kích hoạt cho  $q(1,9)$  không phụ thuộc mẫu tin kích hoạt cho  $r$ . Khi mẫu tin kích hoạt cho  $r$  bị giải phóng thì bộ quản lý Heap có thể dùng vùng nhớ tự do này để cấp phát cho mẫu tin khác. Một số vấn đề thuộc quản lý hiệu quả một Heap sẽ được trình bày trong mục VIII.

## IV. TRUY XUẤT TÊN KHÔNG CỤC BỘ

### 1. Quy tắc tầm vực

Quy tắc tầm vực của ngôn ngữ sẽ xác định việc xử lý khi tham khảo đến các tên không cục bộ.

Quy tắc tầm vực bao gồm hai loại: **Quy tắc tầm tĩnh** và **quy tắc tầm động**.

**Quy tắc tầm tĩnh** (static - scope rule): Xác định sự khai báo áp dụng cho một tên bằng cách kiểm tra văn bản chương trình nguồn. Các ngôn ngữ Pascal, C và Ada sử dụng quy tắc tầm tĩnh với một quy định bổ sung: “tầm gần nhất”.

**Quy tắc tầm động** (dynamic- scope rule): Xác định sự khai báo có thể áp dụng cho một tên tại thời gian thực hiện bằng cách xem xét hoạt động hiện hành. Các ngôn ngữ Lisp, APL và Snobol sử dụng quy tắc tầm động.

### 2. Cấu trúc khối

Một khối bắt đầu bởi một tập hợp các khai báo cho tên (khai báo biến, định nghĩa kiểu, định nghĩa hằng...) sau đó là một tập hợp các lệnh mà trong đó các tên có thể được tham khảo.

Cấu trúc khối thường được sử dụng trong các ngôn ngữ cấu trúc như Pascal, Ada, PL/1. Trong đó chương trình hay chương trình con được tổ chức thành các khối lồng nhau.

Ngôn ngữ cấu trúc khối sử dụng quy tắc tầm tĩnh. Tầm của một khai báo được cho bởi **quy tắc tầm gần nhất** (most closely nested).

1. Một khai báo tại đầu một khối xác định một tên cục bộ trong khối đó. Bất kỳ một tham khảo tới tên trong thân khối được xem xét như là một tham khảo tới dữ liệu cục bộ trong khối nếu nó tồn tại.

2. Nếu một tên  $x$  được tham khảo trong thân một khối  $B$  và  $x$  không được khai báo trong  $B$  thì  $x$  được xem như là một sự tham khảo tới sự khai báo trong  $B'$  là khối nhỏ nhất chứa  $B$ . Nếu trong  $B'$  không có một khai báo cho  $x$  thì lại tham khảo tới  $B''$  là khối nhỏ nhất chứa  $B'$ .

3. Nếu một khối chứa định nghĩa các khối khác thì mọi khai báo trong các khối con hoàn toàn bị che dấu đối với khối ngoài.

Cấu trúc khối có thể cài đặt bằng cách sử dụng cơ chế cấp phát Stack. Khi điều khiển đi vào một khối thì ô nhớ cho các tên được cấp phát và chúng bị thu hồi khi điều khiển rời khỏi khối.

### 3. Tầm tĩnh với các chương trình con không lồng nhau

Quy tắc tầm tĩnh của ngôn ngữ C đơn giản hơn so với Pascal và các định nghĩa chương trình con trong C không lồng nhau. Một chương trình C là một chuỗi các khai báo biến và hàm. Nếu có một sự tham khảo không cục bộ đến tên  $a$  trong một hàm nào đó thì  $a$  phải được tham khảo bên ngoài tất cả các hàm. Tất cả các tên khai báo bên ngoài hàm đều có thể được cấp phát tĩnh. Vị trí các ô nhớ này được biết tại thời gian dịch do đó một tham khảo tới tên không cục bộ trong thân hàm được xác định bằng địa chỉ tuyệt đối. Các tên cục bộ trong hàm nằm trong mẫu tin hoạt động trên đỉnh Stack và có thể xác định bằng cách sử dụng địa chỉ tương đối.

### 4. Tầm tĩnh với các chương trình con lồng nhau.

Trong ngôn ngữ Pascal các chương trình con có thể lồng nhau nhiều cấp.

**Ví dụ 7.5:** Xét chương trình

- (1) **program** sort(input, output);
- (2) var a: array [0...10] of integer;
- (3) x : integer;
- (4) **procedure** readarray;
- (5) var y : integer;
- (6) begin ... a... end; {readarray}
- (7) **procedure** exchange(i,j:integer);
- (8) begin
- (9) x:= a[i]; a[i] := a[j]; a[j] := x;
- (10) end; {exchange}
- (11) **procedure** quicksort(m,n:integer);
- (12) var k,v: integer;

```

(13) function partition(y,z: integer) : integer;
(14)   var i,j : integer;
(15)   begin...a...
(16)       ...v...
(17)       ...exchange(i,j)...
(18)   end; {partition}
(19) begin...end; {quicksort}
(20) begin...end; {sort}

```

**Hình 7.13** - Một chương trình Pascal với các chương trình con lồng nhau

Xét chương trình con partition, trong đó tham khảo đến các tên không cục bộ như:

a: Khai báo trong chương trình chính.

v: khai báo trong quicksort;

exchange:khai báo trong chương trình chính.

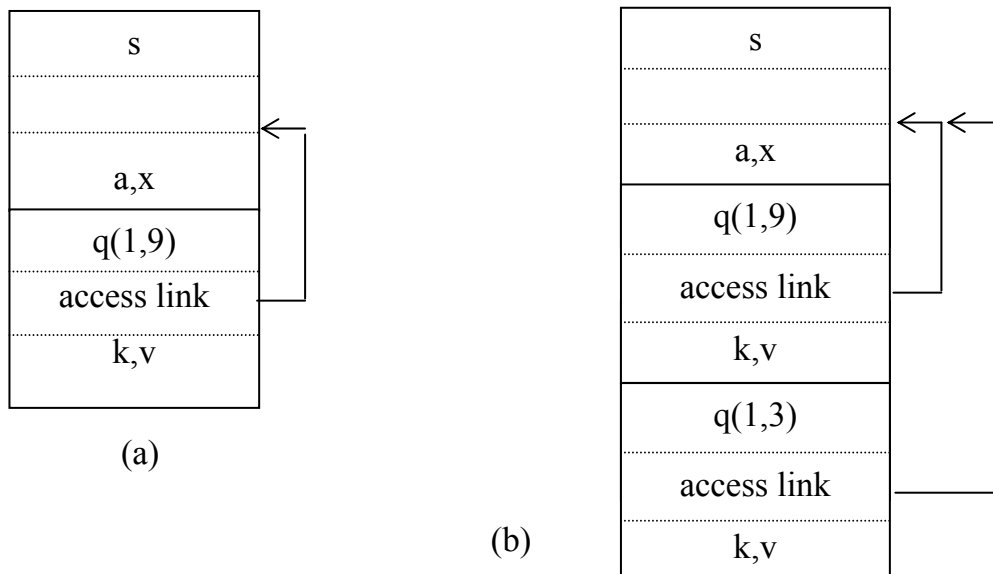
### Độ sâu của sự lồng nhau

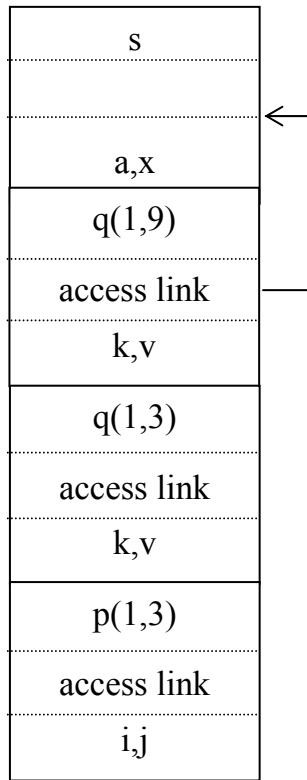
Chúng ta sử dụng thuật ngữ độ lồng sâu để chỉ tầm tĩnh. Tên của chương trình chính có độ sâu cấp một và chúng ta tăng thêm một khi đi từ một chương trình con vào một chương trình con được bao (khai báo) trong nó. Như vậy trong chương trình con partition, a có độ sâu cấp 1, v có độ sâu cấp 2, i có độ sâu cấp 3. Quicksort có độ sâu cấp 2, partition có độ sâu cấp 3, exchange có độ sâu cấp 2.

### Liên kết truy xuất

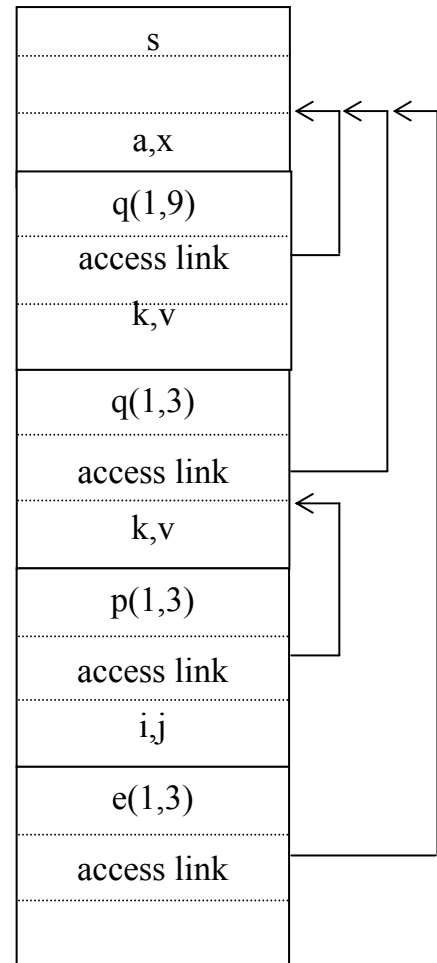
Để cài đặt tầm tĩnh cho các chương trình con lồng nhau ta dùng con trỏ liên kết truy xuất trong mỗi mẫu tin kích hoạt. Nếu chương trình con p được lồng trực tiếp trong q thì liên kết trong mẫu tin kích hoạt của p trỏ tới liên kết truy xuất của mẫu tin kích hoạt hiện hành của q. Hình sau mô tả nội dung Stack trong khi thực hiện chương trình sort trong ví dụ trên

**Ví dụ 7.6:**





(c)



(d)

**Hình 7.14** - Liên kết truy xuất cho phép tìm kiếm ô nhớ của các tên không cục bộ

Liên kết truy xuất của s rỗng vì s không có bao đóng.

Liên kết truy xuất của một mẫu tin kích hoạt của một chương trình con bất kỳ đều trỏ đến mẫu tin kích hoạt của bao đóng của nó.

Giả sử chương trình con p có độ lồng sâu  $np$  tham khảo tới một tên không cục bộ a có độ lồng sâu  $na \leq np$ . Việc tìm đến địa chỉ của a được tiến hành như sau:

- Khi chương trình con p được gọi thì một mẫu tin kích hoạt của p nằm trên đỉnh Stack. Tính giá trị  $np - na$ . Giá trị này được tính tại thời gian dịch.
- Đi xuống  $np - na$  mức theo liên kết truy xuất ta tìm đến được mẫu tin kích hoạt của chương trình con trong đó a được khai báo. Tại đây địa chỉ của a được xác định bằng cách lấy địa chỉ của mẫu tin cộng với độ dời của a (địa chỉ tương đối của a).

**Ví dụ 7.7** (ứng với hình 7.14c) : Hàm partition có độ lồng sâu là  $np = 3$  tham khảo tới biến a có độ lồng sâu  $na = 1$  và biến v có độ lồng sâu  $nv = 2$ .

Để xác định a cần tính  $np - na = 3 - 1 = 2 \Rightarrow$  cần hạ hai cấp

Từ p(1,3) hạ một cấp đến q(1,3) theo liên kết truy xuất.

Từ q(1,3) hạ một cấp đến s theo liên kết truy xuất đến s là nơi a được khai báo.

Để xác định  $v$  cần tính  $np - nv = 3 - 2 = 1 \Rightarrow$  cần hạ một cấp xuống  $q(1,3)$  là nơi  $v$  được khai báo.

Giả sử chương trình con  $p$  có độ lồng sâu  $np$  gọi chương trình con  $e$  ở độ lồng sâu  $ne$ . Đoạn mã để thiết lập liên kết truy xuất phụ thuộc vào việc chương trình được gọi có được định nghĩa trong chương trình gọi hay không?

**Trường hợp 1:**  $np < ne$ : Chương trình con  $e$  có độ lồng sâu lớn hơn chương trình con  $p$  do đó hoặc  $e$  được lồng trong  $p$  hoặc  $p$  không thể tham khảo đến  $e$  ( $e$  bị che dấu khỏi  $p$ ). Ví dụ `sort` gọi `quickort`, `quicksort` gọi `partition`.

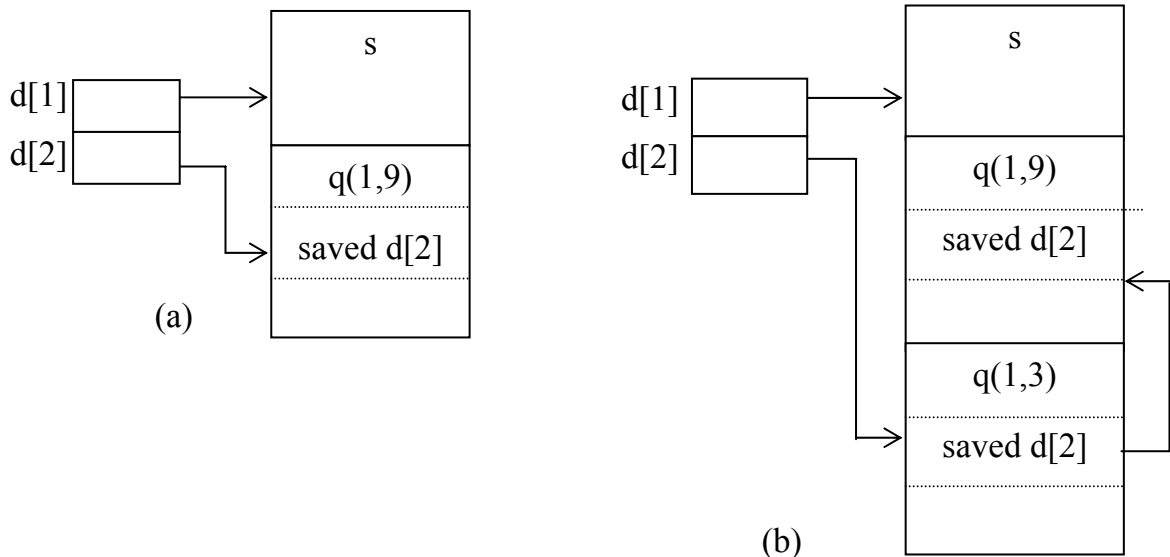
**Trường hợp 2:**  $np \geq ne$ : chương trình con  $e$  có độ lồng sâu nhỏ hơn hoặc bằng độ lồng sâu của chương trình con  $p$ . Theo quy tắc tầm tĩnh thì  $p$  có thể tham khảo  $e$ . Ví dụ `quicksort` gọi chính nó, `partition` gọi `exchange`. Từ chương trình gọi  $np - ne + 1$  bước làm theo liên kết truy nhập ta tìm được mẫu tin kích hoạt của bao đóng gần nhất chứa cả chương trình gọi và chương trình được gọi. Chẳng hạn  $p(1,3)$  gọi  $e(1,3)$ ,  $np = 3$ ,  $ne = 2$ . Ta phải làm  $3 - 2 + 1$  bằng hai bước theo liên kết truy xuất từ  $p$  đến  $s$ .

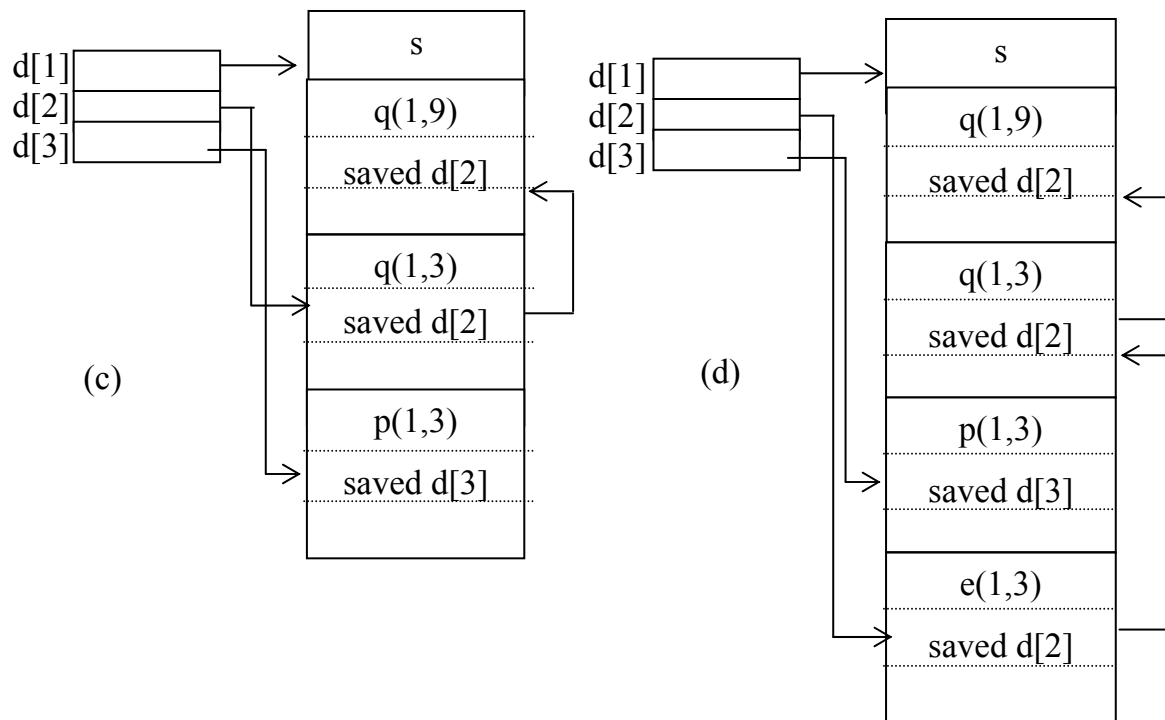
**Display:** để truy xuất nhanh các tên không cục bộ người ta dùng một mảng  $d$  các con trỏ tới các mẫu tin kích hoạt mảng này gọi là `display`.

Giả sử điều khiển nằm trong hoạt động của chương trình con  $t$  có độ lồng sâu  $j$  thì  $j-1$  phần tử của `display` trỏ tới các mẫu tin kích hoạt của các bao đóng gần nhất của  $p$  và  $d[j]$  trỏ tới kích hoạt của  $p$ .

Một tên không cục bộ  $a$  có độ sâu  $i$  nằm trong mẫu tin kích hoạt được trỏ bởi  $d[i]$ .

**Ví dụ 7.8:**





**Hình 7.15** - Sử dụng display khi các chương trình con không được truyền như các tham số

(a): Tình trạng trước khi  $q(1,3)$  bắt đầu, quicksort có độ lồng sâu cấp 2,  $d[2]$  được gửi cho mẫu tin kích hoạt của quicksort khi nó bắt đầu. Giá trị của  $d[2]$  được lưu trong mẫu tin kích hoạt của  $q(1,9)$ .

(b): Khi  $q(1,3)$  bắt đầu  $d[2]$  trở tới mẫu tin kích hoạt mức ứng với  $q(1,3)$ , giá trị của  $d[2]$  lại được lưu trong mẫu tin này. Giá trị này là cần thiết để phục hồi display cũ khi điều khiển trả về cho  $q(1,9)$ . Như vậy khi một mẫu tin kích hoạt mới được đẩy vào Stack thì:

- Lưu giá trị của  $d[i]$  vào mẫu tin đó.
- Đặt  $d[i]$  trở tới mẫu tin đó.

Khi một mẫu tin được pop khỏi Stack thì  $d[i]$  được phục hồi.

Giả sử một chương trình con có độ lồng sâu cấp  $j$  gọi một chương trình con có độ lồng sâu cấp  $i$ . Có hai trường hợp xảy ra phụ thuộc chương trình con được gọi có được định nghĩa trong chương trình gọi hay không.

**Trường hợp 1:**  $j < i \Rightarrow i = j+1$ : thêm ô nhớ  $d[i]$ , cấp phát mẫu tin kích hoạt cho chương trình con  $i$ , ghi  $d[i]$  vào trong đó và đặt  $d[i]$  trở tới nó (ví dụ 7.8a, 7.8c)

**Trường hợp 2:**  $j \geq i$ : Ghi giá trị cũ của  $d[i]$  vào mẫu tin kích hoạt mới và đặt  $d[i]$  trở vào mẫu tin cuối. (ví dụ 7.8b và 7.8d)

## 5. Tầm động

Với khái niệm tầm động, một hoạt động mới kế thừa sự liên kết đã tồn tại của một tên không cục bộ. Tên không cục bộ  $a$  trong hoạt động của chương trình được gọi tham khảo đến cùng một ô nhớ như trong hoạt động của chương trình gọi. Đối với tên cục bộ thì một liên kết mới được thiết lập tới ô nhớ trong mẫu tin hoạt động mới.



**Ví dụ 7.9:** Xét chương trình:

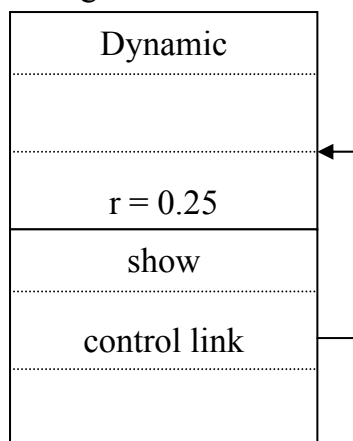
```
(1)  program  dynamic (input, output);  
(2)    var r : real;  
(3)  procedure show;  
(4)    begin write(r : 5 : 3); end;  
(5)  procedure small;  
(6)    var r : real;  
(7)    begin r := 0.125; show; end;  
(8)  begin  
(9)    r := 0.25;  
(10)  show, small, writeln;  
(11) end;
```

**Hình 7.16 - Kết quả chương trình tùy thuộc vào tầm động hay tầm tĩnh được sử dụng**

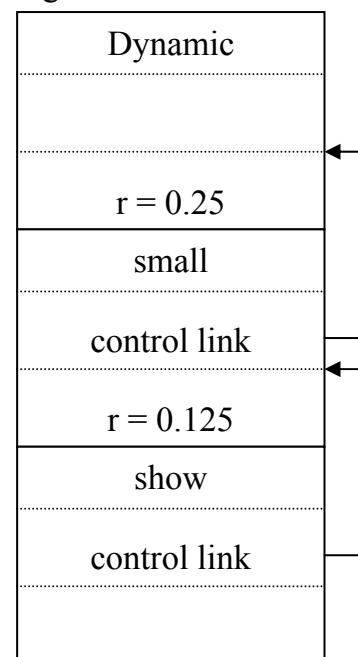
Kết quả thực hiện chương trình:

- Dưới tầm tĩnh;  
0.250 0.250
- Dưới tầm động:  
0.250 0.125

Khi show được gọi tại dòng 10 trong chương trình chính thì 0.250 được in ra vì r của chương trình chính được sử dụng. Tuy nhiên khi show được gọi tại dòng 7 trong small thì 0.125 được in ra vì r của chương trình con small được sử dụng. Cơ chế tầm động sử dụng liên kết điều khiển để tham khảo tên không cục bộ.



Show được gọi tại dòng 10  
tham khảo r= 0.25



Show được gọi tại dòng 7  
tham khảo r = 0.125

**Hình 7.17** - Sử dụng liên kết điều khiển để tham khảo các tên không cục bộ

## V. TRUYỀN THAM SỐ

Khi một chương trình con gọi một chương trình con khác thì phương pháp thông thường để giao tiếp giữa chúng là thông qua tên không cục bộ và thông qua các tham số của chương trình được gọi.

**Ví dụ 7.10:** Để đổi hai giá trị  $a[i]$  và  $a[j]$  cho nhau ta dùng

- (1) **procedure** exchange( $i, j$  : integer);
- (2) var  $x$  : integer;
- (3) begin
- (4)  $x := a[i]$ ;  $a[i] := a[j]$ ;  $a[j] := x$ ;
- (5) end;

trong đó mảng  $a$  là tên không cục bộ và  $i, j$  là các tham số.

Có rất nhiều phương pháp truyền tham số như:

- Truyền bằng giá trị (Transmission by value, call- by-value)
- Truyền bằng tham khảo (Transmission by name, call- by-name)...

Ở đây chúng ta xét hai phương pháp phổ biến nhất:

### 1. Truyền bằng giá trị

Là phương pháp đơn giản nhất của truyền tham số được sử dụng trong C và Pascal. Truyền bằng giá trị được xử lý như sau:

1. Tham số hình thức được xem như là tên cục bộ do đó ô nhớ của các tham số hình thức nằm trong mẫu tin kích hoạt của chương trình được gọi.
2. Chương trình gọi đánh giá các tham số thực tế và đặt các giá trị của chúng vào trong ô nhớ của tham số hình thức.

### 2. Truyền tham chiếu (truyền địa chỉ hay truyền vị trí)

Chương trình gọi truyền cho chương trình được gọi con trỏ tới địa chỉ của mỗi một tham số thực tế.

**Ví dụ 7.11:**

- (1) **program** reference (input, output)
- (2) var  $i$ : integer;
- (3)  $a$ : array[0...10] of integer;
- (4) **procedure** swap(var  $x, y$ : integer);
- (5) var  $temp$  : integer;
- (6) begin
- (7)  $temp := x$ ;

```

(8)    x := y;
(9)    y := temp;
(10)   end;
(11)   begin
(12)    i := 1;  a[1] := 2;
(13)    swap(i,a[1]);
(14)   end;

```

**Hình 7.18** - Chương trình Pascal với thủ tục swap

Với lời gọi tại dòng (13) ta có các bước sau:

1. Copy địa chỉ của i và a[i] vào trong mẫu tin hoạt động của swap thành arg1, arg2 tương ứng với x, y.
2. Đặt temp bằng nội dung của vị trí được trả về bởi arg1 tức là temp := 1. Bước này tương ứng lệnh temp := x trong dòng (7) của swap.
3. Đặt nội dung của vị trí được trả về bởi arg1 bởi giá trị của vị trí được trả về bởi arg2, tức là i := a[1]. Bước này tương ứng lệnh x := y trong dòng (8) của swap.
4. Đặt nội dung của vị trí được trả về bởi arg2 bởi giá trị của temp. Tức là a[1] := i. Bước này tương ứng lệnh y := temp.

## VI. BẢNG KÝ HIỆU

Chương trình dịch sẽ sử dụng bảng ký hiệu để lưu trữ thông tin về tầm vực và mối liên kết của các tên. Bảng ký hiệu được truy xuất nhiều lần mỗi khi một tên xuất hiện trong chương trình nguồn.

Có hai cơ chế tổ chức bảng ký hiệu là danh sách tuyến tính và bảng băm.

### 1. Cấu trúc một ô của bảng ký hiệu

Mỗi ô trong bảng ký hiệu tương ứng với một tên. Định dạng của các ô này thường không giống nhau vì thông tin lưu trữ về một tên phụ thuộc vào việc sử dụng tên đó. Thông thường một ô được cài đặt bởi một mẫu tin. Nếu muốn có được sự đồng nhất của các mẫu tin ta có thể lưu thông tin bên ngoài bảng ký hiệu, trong mỗi ô của bảng chỉ chứa các con trỏ trỏ tới thông tin đó,

Trong bảng ký hiệu cũng có thể có lưu các từ khóa của ngôn ngữ. Nếu vậy thì chúng phải được đưa vào bảng ký hiệu trước khi bộ phân tích từ vựng bắt đầu.

### 2. Vấn đề lưu trữ lexeme của danh biểu

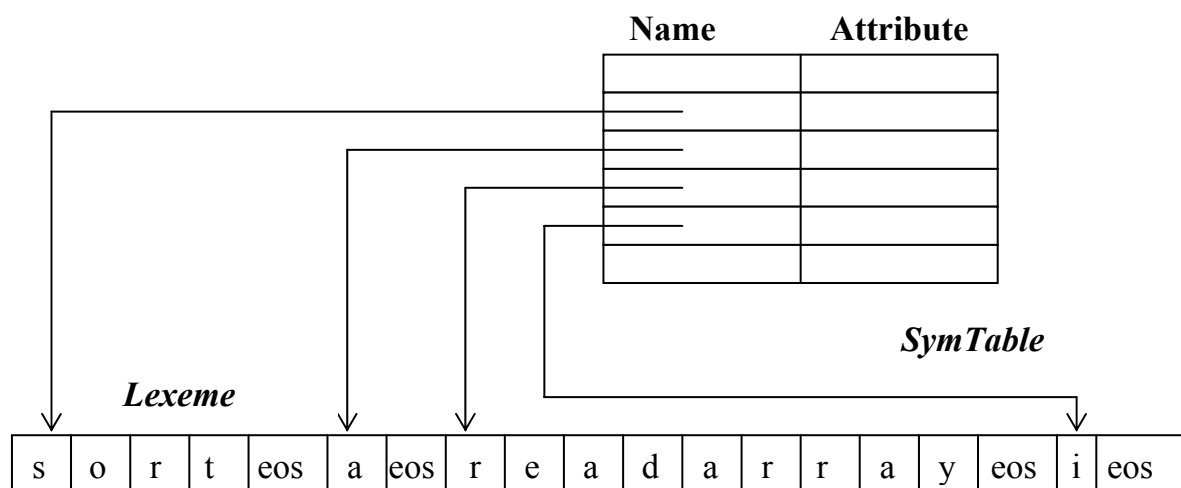
Các danh biểu trong các ngôn ngữ lập trình thường có hai loại: Một số ngôn ngữ quy định độ dài của danh biểu không được vượt quá một giới hạn nào đó. Một số khác không giới hạn về độ dài.

Trường hợp danh biểu bị giới hạn về độ dài thì chuỗi các ký tự tạo nên danh biểu được lưu trữ trong bảng ký hiệu.

Name									Attribute
s	o	r	t						
a									
r	e	a	d	a	r	r	a	y	
i									

**Hình 7.19** - Bảng ký hiệu lưu giữ các tên bị giới hạn độ dài

Trường hợp độ dài tên không bị giới hạn thì các Lexeme được lưu trong một mảng riêng và bảng ký hiệu chỉ giữ các con trỏ trỏ tới đầu mỗi Lexeme



**Hình 7.20** - Bảng ký hiệu lưu giữ các tên không bị giới hạn độ dài

### 3. Tổ chức bảng ký hiệu bằng danh sách tuyến tính

Cấu trúc đơn giản, dễ cài đặt nhất cho bảng ký hiệu là danh sách tuyến tính của các mẫu tin. Ta dùng một mảng hoặc nhiều mảng tương đương để lưu trữ tên và các thông tin kết hợp với chúng. Các tên mới được đưa vào trong danh sách theo thứ tự mà chúng được phát hiện. Vị trí của mảng được đánh dấu bởi con trỏ available chỉ ra một ô mới của bảng sẽ được tạo ra.

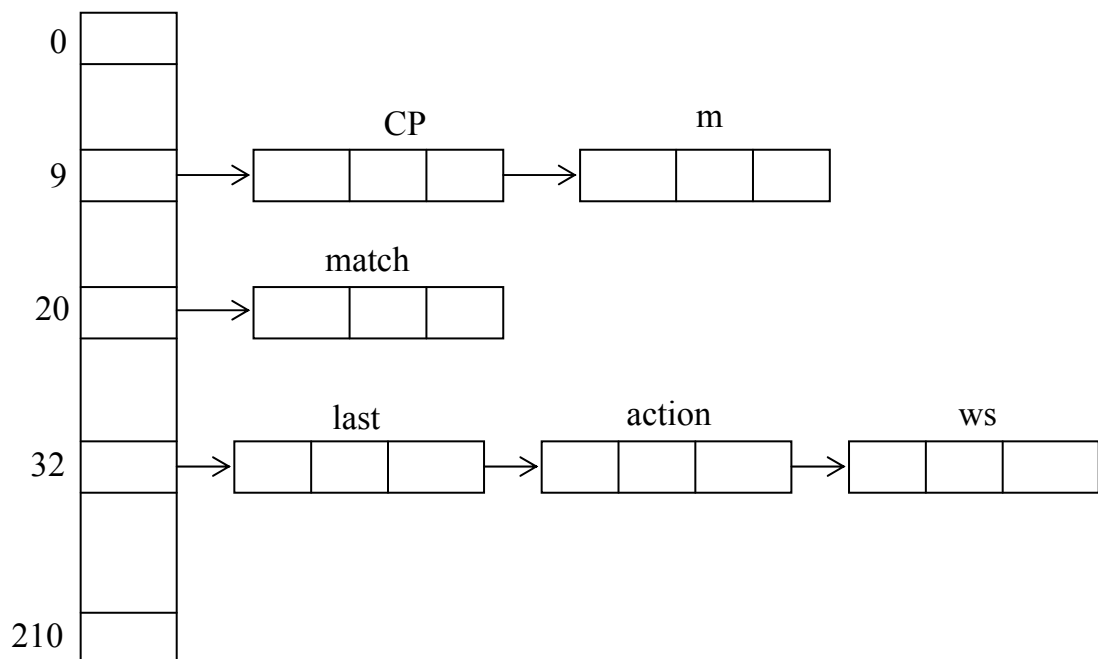
Việc tìm kiếm một tên trong bảng ký hiệu được bắt đầu từ available đến đầu bảng. Trong các ngôn ngữ cấu trúc khối sử dụng quy tắc tầm tĩnh. Thông tin kết hợp với tên có thể bao gồm cả thông tin về độ sâu của tên. Bằng cách tìm kiếm từ available trở về đầu mảng chúng ta đảm bảo rằng sẽ tìm thấy tên trong tầng gần nhất.

id1
info 1
id2
info2
...

**Hình 7.21 - Danh sách tuyến tính các mẫu tin**

#### 4. Tổ chức bảng ký hiệu bằng bảng băm

Kỹ thuật sử dụng bảng băm để cài đặt bảng ký hiệu thường được sử dụng vì tính hiệu quả của nó. Cấu tạo bao gồm hai phần; bảng băm và các danh sách liên kết.



**Hình 7.22 - Bảng băm có kích thước 211**

1. Bảng băm là một mảng bao gồm m con trỏ.

2. Bảng danh biểu được chia thành m danh sách liên kết, mỗi danh sách liên kết được trỏ bởi một phần tử trong bảng băm.

Việc phân bổ các danh biểu vào danh sách liên kết nào do hàm băm (hash function) quy định. Giả sử s là chuỗi ký tự xác định danh biểu, hàm băm h tác động lên s trả về một giá trị nằm giữa 0 và m- 1  $h(s) = t \Rightarrow$  Danh biểu s được đưa vào trong danh sách liên kết được trỏ bởi phần tử t của bảng băm.

Có nhiều phương pháp để xác định hàm băm.

Phương pháp đơn giản nhất như sau:

1. Giả sử s bao gồm các ký tự  $c_1, c_2, c_3, \dots, c_k$ . Mỗi ký tự cho ứng với một số nguyên dương  $n_1, n_2, n_3, \dots, n_k$ ; lấy  $h = n_1 + n_2 + \dots + n_k$ .
2. Xác định  $h(s) = h \bmod m$

## BÀI TẬP CHƯƠNG VII

**7.1.** Hãy dùng quy tắc tầm vực của ngôn ngữ Pascal để xác định tầm vực ý nghĩa của các khai báo cho mỗi lần xuất hiện tên a, b trong chương trình sau. Output của chương trình là các số nguyên từ 1 đến 4.

```
Program    a ( input, output);  
Procedure  b ( u, v, x, y : integer);  
    Var      a : record    a, b : integer end;  
            b : record    a, b : integer end;  
  
    begin  
        With a do begin    a := u ; b := v  end;  
        With b do begin    a := x ; b := y  end;  
        Writeln ( a.a, a.b, b.a, b.b );  
  
    end;  
  
Begin  
    B ( 1, 2, 3, 4)  
  
End.
```

**7.2.** Chương trình sau sẽ in ra giá trị như thế nào nếu giả sử thông số được truyền bằng:

- a) trị
- b) quy chiếu
- c) trị - kết quả
- d) tên

```
Program    main ( input, output);  
Procedure  p ( x, y, z );  
    begin  
        y := y + 1;  
        z := z + x;  
  
    end;  
  
Begin  
    a := 2 ;  
    b := 3 ;  
    p ( a +b ; a, a )  
    print a
```

**End.**

**7.3.** Cho đoạn chương trình trong **Algol** như sau :

```
begin ...  
Procedure A ( px); procedure px      { tham số hình thức px là thủ tục }  
begin  
    procedure B ( pz); procedure pz    { tham số hình thức pz là thủ tục }  
        begin  
            ....  
            pz;  
            ....  
        end;  
    B (px);  
end;  
procedure C;  
    begin  
        procedure D;  
            begin ... end;  
        A(D);  
    end;  
C;  
end.
```

Hãy giải thích quá trình thực thi của chương trình trên, các bước truyền tham số (giải thích bằng hình ảnh của Stack).

**7.4.** Cho đoạn chương trình sau:

```
var a, b : integer;  
Procedure AB  
    Var a, c : real;  
        k, l : integer;  
    procedure AC  
        Var x, y : real;  
            b : array [ 1 .. 10] of integer;  
    begin
```

```

....
    end;
begin
....
    end;
begin
....
    end.

```

Hãy xây dựng bảng ký hiệu thao các phương pháp sau:

a) Danh sách tuyến tính

b) Băm (hash), nếu giả sử ta có kết quả của hàm biến đổi băm như sau:

```

a = 3;      b = 4;      c = 4;      k = 2;      l = 3;
x = 4;      y = 5;      AB = 2;     AC = 6;

```

7.5. Cho đoạn chương trình sau:

```

Program    baitap;
Var    a : real;

    procedure    sub1 ;
        Var    x, y : real;
        begin
            ....
        end;

    procedure    sub2 (t :integer);
        Var    k : integer;

        procedure    sub3 ;
            Var    m : real;
            begin
                ....
            end;

        procedure    t;
            Var    x, y : real;
            begin
                ....
            end;

```



*begin*

....

*end.*

Hãy vẽ bảng ký hiệu cho từng chương trình con có con trỏ trỏ đến bảng ký hiệu của chương trình bị gọi và có con trỏ trỏ ngược lại bảng ký hiệu của chương trình gọi nó.

## CHƯƠNG VIII

### SINH MÃ TRUNG GIAN

#### Nội dung chính:

Thay vì một chương trình nguồn được dịch trực tiếp sang mã đích, nó nên được dịch sang dạng *mã trung gian* bởi kỳ trước trước khi được tiếp tục dịch sang mã đích bởi kỳ sau vì một số tiện ích: Thuận tiện khi muốn thay đổi cách biểu diễn chương trình đích; Giảm thời gian thực thi chương trình đích vì mã trung gian có thể được tối ưu. Chương này giới thiệu *các dạng biểu diễn trung gian* đặc biệt là dạng *mã ba địa chỉ*. Phần lớn nội dung của chương tập trung vào trình bày cách tạo ra một *bộ sinh mã trung gian* đơn giản dạng mã 3 đại chỉ. Bộ sinh mã này dùng phương thức trực tiếp cú pháp để dịch các khai báo, câu lệnh gán, các lệnh điều khiển sang mã ba địa chỉ.

#### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải nắm được cách tạo ra một bộ sinh mã trung gian cho một ngôn ngữ lập trình đơn giản (chỉ chứa một số loại khai báo, lệnh điều khiển và câu lệnh gán) từ đó có thể mở rộng để cài đặt bộ sinh mã cho những ngôn ngữ phức tạp hơn.

#### Tài liệu tham khảo:

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

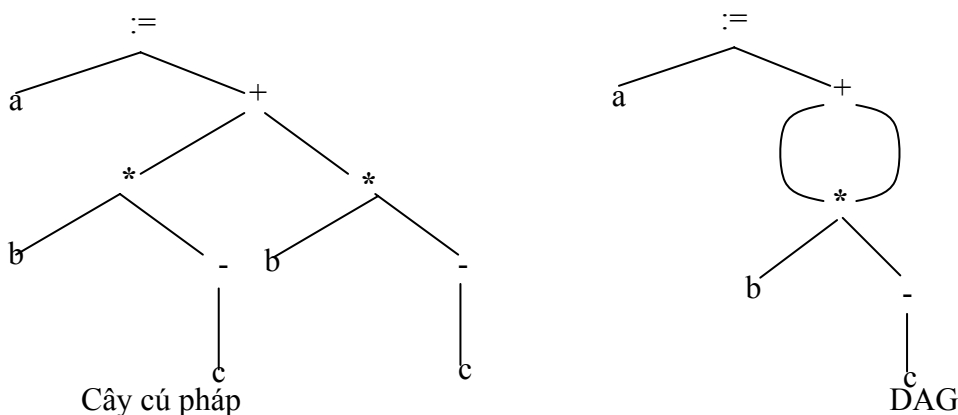
## I. NGÔN NGỮ TRUNG GIAN

Cây cú pháp, ký pháp hậu tố và mã 3 địa chỉ là các loại biểu diễn trung gian.

### 1. Biểu diễn đồ thị

Cây cú pháp mô tả cấu trúc phân cấp tự nhiên của chương trình nguồn. DAG cho ta cùng lượng thông tin nhưng bằng cách biểu diễn ngắn gọn hơn trong đó các biểu thức con không được biểu diễn lặp lại.

**Ví dụ 8.1:** Với lệnh gán  $a := b * - c + b * - c$ , ta có cây cú pháp và DAG:



**Hình 8.1-** Biểu diễn đồ thị của  $a := b * - c + b * - c$

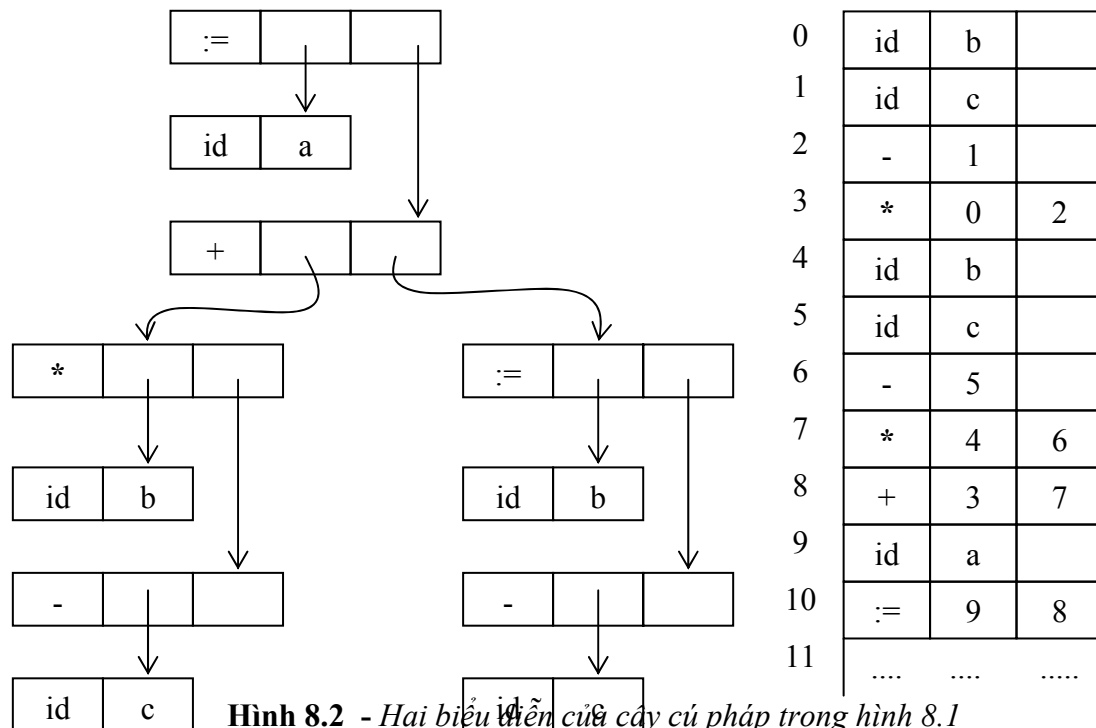
Ký pháp hậu tố là một biểu diễn tuyến tính của cây cú pháp. Nó là một danh sách các nút của cây, trong đó một nút xuất hiện ngay sau con của nó.

$a\ b\ c\ -\ *\ b\ c\ -\ * + :=$  là biểu diễn hậu tố của cây cú pháp hình trên.

Cây cú pháp có thể được cài đặt bằng một trong 2 phương pháp:

- Mỗi nút được biểu diễn bởi một mẫu tin, với một trường cho toán tử và các trường khác trỏ đến con của nó.
- Một mảng các mẫu tin, trong đó chỉ số của phần tử mảng đóng vai trò như là con trỏ của một nút.

Tất cả các nút trên cây cú pháp có thể tuân theo con trỏ, bắt đầu từ nút gốc tại 10



Hình 8.2 - Hai biểu diễn của cây cú pháp trong hình 8.1

## 2. Mã 3 địa chỉ

Mã lệnh 3 địa chỉ là một chuỗi các lệnh có dạng tổng quát là  $x := y \text{ op } z$ . Trong đó  $x, y, z$  là tên, hằng hoặc dữ liệu tạm sinh ra trong khi dịch,  $op$  là một toán tử số học hoặc logic.

Chú ý rằng không được có quá một toán tử ở vế phải của mỗi lệnh. Do đó biểu thức  $x + y * z$  phải được dịch thành chuỗi :

$t1 := y * z$

$t2 := x + t1$

Trong đó  $t1, t2$  là những tên tạm sinh ra trong khi dịch.

Mã lệnh 3 địa chỉ là một biểu diễn tuyến tính của cây cú pháp hoặc DAG, trong đó các tên tương minh biểu diễn cho các nút trong trên đồ thị.

$t1 := -c$

$t2 := b * t1$

$t3 := -c$

$t4 := b * t3$

$t1 := -c$

$t2 := b * t1$

$t3 := t2 + t2$

$a := t3$

$$t5 := t2 + t4$$

$$a := t5$$

Mã lệnh 3 địa chỉ của cây cú pháp    Mã lệnh 3 địa chỉ của DAG

**Hình 8.3** - Mã lệnh 3 địa chỉ tương ứng với cây cú pháp và DAG trong hình 8.1

### 3. Các mã lệnh 3 địa chỉ phổ biến

1. Lệnh gán dạng **x := y op z**, trong đó op là toán tử 2 ngôi số học hoặc logic.
2. Lệnh gán dạng **x := op y**, trong đó op là toán tử một ngôi. Chẳng hạn, phép lấy số đối, toán tử NOT, các toán tử SHIFT, các toán tử chuyển đổi.
3. Lệnh COPY dạng **x := y**, trong đó giá trị của y được gán cho x.
4. Lệnh nhảy không điều kiện **goto L**. Lệnh 3 địa chỉ có nhãn L là lệnh tiếp theo thực hiện.
5. Các lệnh nhảy có điều kiện như **if x relop y goto L**. Lệnh này áp dụng toán tử quan hệ relop (<, =, >=, .. ..) vào x và y. Nếu x và y thỏa quan hệ relop thì lệnh nhảy với nhãn L sẽ được thực hiện, ngược lại lệnh đứng sau IF x relop y goto L sẽ được thực hiện.
6. **param x và call p, n** cho lời gọi chương trình con và **return y**. Trong đó, y biểu diễn giá trị trả về có thể lựa chọn. Cách sử dụng phổ biến là một chuỗi lệnh 3 địa chỉ.

*param     x1*

*param     x2*

*.. .. .*

*param     xn*

*call                     p, n*

được sinh ra như là một phần của lời gọi chương trình con p (x1,x2,.. .., xn).

7. Lệnh gán dạng **x := y[i] và x[i] := y**. Lệnh đầu lấy giá trị của vị trí nhớ của y được xác định bởi giá trị ô nhớ i gán cho x. Lệnh thứ 2 lấy giá trị của y gán cho ô nhớ x được xác định bởi i.
8. Lệnh gán địa chỉ và con trỏ dạng **x := &y, x := \*y và \*x := y**. Trong đó, x := &y đặt giá trị của x bởi vị trí của y. Câu lệnh x := \*y với y là một con trỏ mà r\_value của nó là một vị trí, r\_value của x đặt bằng nội dung của vị trí này. Cuối cùng \*x := y đặt r\_value của đối tượng được trỏ bởi x bằng r\_value của y.

### 4. Dịch trực tiếp cú pháp thành mã lệnh 3 địa chỉ

**Ví dụ 8.2:** Định nghĩa S<sub>2</sub> thuộc tính sinh mã lệnh địa chỉ cho lệnh gán:

Luật sinh	Luật ngữ nghĩa
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place := E_1.place '+' E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$

	$gen(E.place := E_1.place * E_2.place)$ $E.place := newtemp;$ $E.code := E_1.code    gen(E.place := 'uminus' E_1.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E.place := id.place;$ $E.code := ''$
$E \rightarrow id$	

**Hình 8.4 - Định nghĩa trực tiếp cú pháp sinh mã lệnh ba địa chỉ cho lệnh gán**

Với chuỗi nhập  $a = b * -c + b * -c$ , nó sinh ra mã lệnh 3 địa chỉ

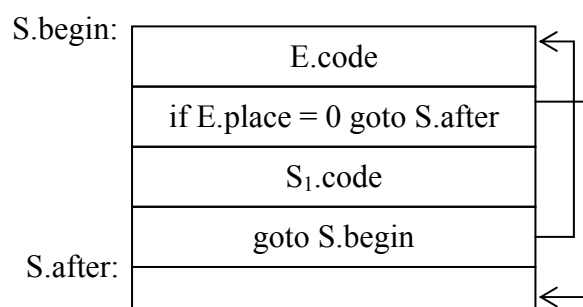
$t1 := -c$   
 $t2 := b * t1$   
 $t3 := -c$   
 $t4 := b * t3$   
 $t5 := t2 + t4$   
 $a := t5$

thuộc tính tổng hợp S.code biểu diễn mã 3 địa chỉ cho lệnh gán S. Ký hiệu chưa kết thúc E có 2 thuộc tính E.place là giá trị của E và E.code là chuỗi lệnh 3 địa chỉ để đánh giá E

Khi mã lệnh 3 địa chỉ được sinh, tên tạm được tạo ra cho mỗi nút trong trên cây cú pháp.

Giá trị của ký hiệu chưa kết thúc E trong luật sinh  $E \rightarrow E_1 + E_2$  được tính vào trong tên tạm t. Nói chung mã lệnh 3 địa chỉ cho lệnh gán  $id := E$  bao gồm mã lệnh cho việc đánh giá E vào trong biến tạm t, sau đó là một lệnh gán  $id.place := t$ .

Hàm newtemp trả về một chuỗi các tên  $t1, t2, \dots, tn$  tương ứng các lời gọi liên tiếp. Gen ( $x := y + z$ ) để biểu diễn lệnh 3 địa chỉ  $x := y + z$



Luật sinh	Luật ngữ nghĩa
$S \rightarrow \text{while } E \text{ do } S_1$	$S.begin := newlabel;$ $S.after := newlabel;$ $S.code := gen(S.begin ':')    E.code   $ $gen('if' E.place '=' 0 'goto' S.after)   $ $S_1.code    gen('goto' S.begin)    gen(S.after ':')$

### Hình 8.5 - Định nghĩa trực tiếp cú pháp sinh mã lệnh ba địa chỉ cho câu lệnh while

Lệnh  $S \rightarrow \text{while } E \text{ do } S1$  được sinh ra bằng cách dùng các thuộc tính  $S.\text{begin}$  và  $S.\text{after}$  để đánh dấu lệnh đầu tiên trong đoạn mã lệnh của  $E$  và lệnh sau đoạn mã lệnh của  $S$ .

Hàm newlabel trả về một nhãn mới tại mỗi lần được gọi

## 5. Cài đặt lệnh 3 địa chỉ

Lệnh 3 địa chỉ là một dạng trừu tượng của mã lệnh trung gian. Trong chương trình dịch, các mã lệnh này có thể cài đặt bằng một mẫu tin với các trường cho toán tử và toán hạng. Có 3 cách biểu diễn là bộ tứ, bộ tam và bộ tam gián tiếp.

### ▪ Bộ tứ

Bộ tứ (quadruples) là một cấu trúc mẫu tin có 4 trường ta gọi là  $op$ ,  $arg1$ ,  $arg2$  và  $result$ . Trường  $op$  chứa toán tử. Lệnh 3 địa chỉ  $x := y \text{ op } z$  được biểu diễn bằng cách thay thế  $y$  bởi  $arg1$ ,  $z$  bởi  $arg2$  và  $x$  bởi  $result$ . Các lệnh với toán tử một ngôi như  $x := -y$  hay  $x := y$  thì không sử dụng  $arg2$ . Các toán tử như  $param$  không sử dụng cả  $arg2$  lẫn  $result$ . Các lệnh nhảy có điều kiện và không điều kiện đặt nhãn đích trong  $result$ .

Nội dung các trường  $arg1$ ,  $arg2$  và  $result$  trở tới ô trong bảng ký hiệu đối với các tên biểu diễn bởi các trường này. Nếu vậy thì các tên tạm phải được đưa vào bảng ký hiệu khi chúng được tạo ra.

**Ví dụ 8.3:** Bộ tứ cho lệnh  $a := b * -c + b * -c$

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Hình 8.6 - Biểu diễn bộ tứ cho các lệnh ba địa chỉ

### ▪ Bộ tam

Để tránh phải lưu tên tạm trong bảng ký hiệu; chúng ta có thể tham khảo tới giá trị tạm bằng vị trí của lệnh tính ra nó. Để làm điều đó ta sử dụng bộ tam (triples) là một mẫu tin có 3 trường  $op$ ,  $arg1$  và  $arg2$ . Trong đó,  $arg1$  và  $arg2$  trở tới bảng ký hiệu (đối với tên hoặc hằng do người lập trình định nghĩa) hoặc trở tới một phần tử trong bộ tam (đối với giá trị tạm)

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Hình 8.7 - Biểu diễn bộ tam cho các lệnh ba địa chỉ

Các lệnh như  $x[i] := y$  và  $x := y[i]$  sử dụng 2 ô trong cấu trúc bộ tam.

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

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

**Hình 8.8** - Biểu diễn bộ tam cho  $x[i] := y$  và  $x := y[i]$

#### ▪ Bộ tam gián tiếp

Một cách biểu diễn khác của bộ tam là thay vì liệt kê các bộ tam trực tiếp ta sử dụng một danh sách các con trỏ các bộ tam.

	statements
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

**Hình 8.9** - Biểu diễn bộ tam gián tiếp cho các lệnh ba địa chỉ

## II. KHAI BÁO

### 1. Khai báo trong chương trình con

Các tên cục bộ trong chương trình con được truy xuất đến thông qua địa chỉ tương đối của nó. Gọi là offset.

**Ví dụ 8.4:** Xét lược đồ dịch cho việc khai báo biến

$P \rightarrow \{offset := 0\} D$

$D \rightarrow D ; D$

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

$T \rightarrow integer \quad \{T.type := integer; T.width := 4\}$

$T \rightarrow real \quad \{T.type := real; T.width := 8\}$

$T \rightarrow array[num] \text{ of } T1 \quad \{T.type := array(num.val, T1.type);$   
 $T.width := num.val * T1.width\}$

$T \rightarrow \uparrow T1 \quad \{T.type := pointer(T1.type); T.width := 4\}$

**Hình 8.10** - Xác định kiểu và địa chỉ tương đối của các tên được khai báo

Trong ví dụ trên, ký hiệu chưa kết thúc P sinh ra một chuỗi các khai báo dạng id:T.

Trước khi khai báo đầu tiên được xét thì offset = 0. Khi mỗi khai báo được tìm thấy tên và giá trị của offset hiện tại được đưa vào trong bảng ký hiệu, sau đó offset được tăng lên một khoảng bằng kích thước của đối tượng dữ liệu được cho bởi tên đó.

Thủ tục **enter(name, type, offset)** tạo một ô trong bảng ký hiệu với tên, kiểu và địa chỉ tương đối của vùng dữ liệu của nó. Ta sử dụng các thuộc tính tổng hợp type và width để chỉ ra kiểu và kích thước (số đơn vị nhớ) của kiểu đó.

Chú ý rằng lược đồ dịch  $P \rightarrow \{offset := 0\} D$  có thể được viết lại bằng cách thay thế hành vi  $\{offset := 0\}$  bởi một ký hiệu chưa kết thúc M để được

$P \rightarrow MD$

$M \rightarrow \varepsilon \{offset := 0\}$

Tất cả các hành vi đều nằm cuối về phải.

## 2. Lưu trữ thông tin về tầm

Trong một ngôn ngữ mà chương trình con được phép khai báo lồng nhau. Khi một chương trình con được tìm thấy thì quá trình khai báo của chương trình con bao bị tạm dừng.

Vấn phạm cho sự khai báo đó là;

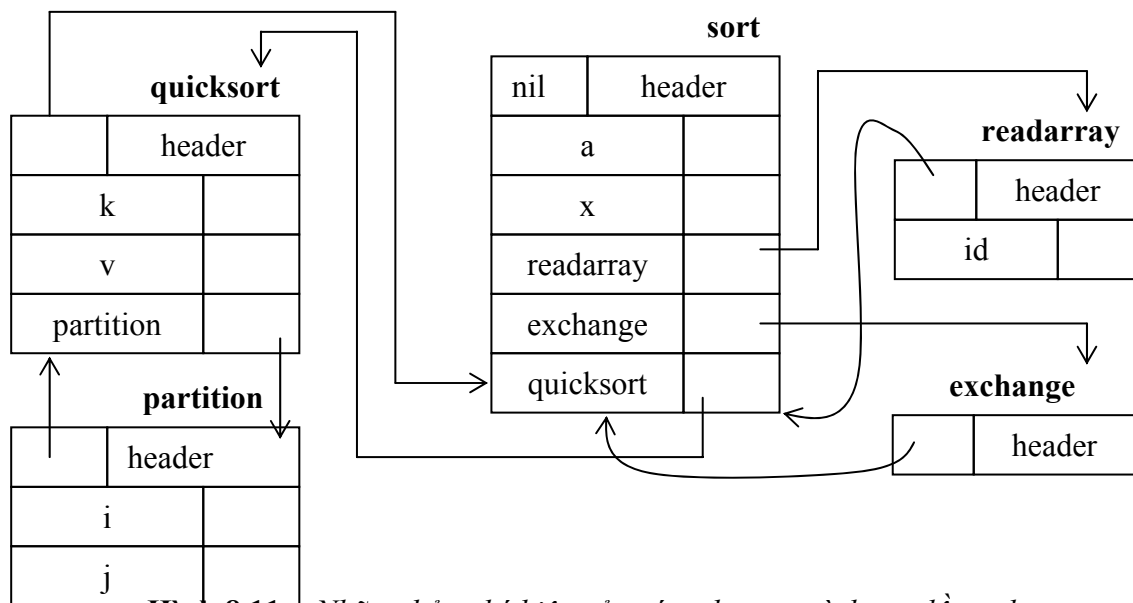
$P \rightarrow D$

$D \rightarrow D ; D \mid id: T \mid proc\ id ; D; S$

Để đơn giản chúng ta tạo ra một bảng ký hiệu riêng cho mỗi chương trình con.

Khi một khai báo chương trình con  $D \rightarrow proc\ id\ D1 ; S$  được tạo ra và các khai báo trong  $D1$  được lưu trữ trong bảng ký hiệu mới.

**Ví dụ 8.5:** Chương trình Sort có bốn chương trình con lồng nhau readarray, exchange, quicksort và partition. Ta có năm bảng ký hiệu tương ứng.



**Hình 8.11** - Những bảng ký hiệu của các chương trình con lồng nhau

**Luật ngữ nghĩa được xác định bởi các thao tác sau**

1. *mktable (previous)*: Tạo một bảng ký hiệu mới và con trỏ tới bảng đó. Tham số previous là một con trỏ tới bảng ký hiệu của chương trình con bao. Con trỏ previous được lưu trong header của bảng ký hiệu mới. Trong header còn có thể có các thông tin khác như độ sâu lồng của chương trình con.
2. *enter (table, name, type, offset)*: Tạo một ô mới trong bảng ký hiệu được trỏ bởi table.
3. *addwidth (table, width)*: Ghi kích thước tích lũy của tất cả các ô trong bảng vào trong header kết hợp với bảng đó.



4. *enterproc (table, name, newtable)*: Tạo một ô mới cho tên chương trình con vào trong bảng được trỏ bởi table. newtable trỏ tới bảng ký hiệu của chương trình con này.

Ta có lược đồ dịch

$P \rightarrow M D$	$\{ addwidth(top(tblptr), top(offset));$ $pop(tblptr); pop(offset) \}$
$M \rightarrow \varepsilon$	$\{ t:=mktable(nil); push(t, tblptr) ; push(0,offset) \}$
$D \rightarrow D_1 ; D_2$	
$D \rightarrow proc\ id ; N D_1 ; S$	$\{ t:= top(tblptr); addwidth(t, top(offset)); pop(tblptr);$ $pop(offset); enterproc(top(tblptr), id\_name,t) \}$
$D \rightarrow id : T$	$\{ enter(top(tblptr), id\_name, T.type, top(offset));$ $top(offset):= top(offset) + T.width \}$
$N \rightarrow \varepsilon$	$\{ t:=mktable(top(tblptr)); push(t, tblptr);$ $push(0,offset) \}$

**Hình 8.12** - Xử lý các khai báo trong những chương trình con lồng nhau

Ta dùng Stack **tblptr** để giữ các con trỏ bảng ký hiệu.

Chẳng hạn, khi các khai báo của partition được khảo sát thì trong tblptr chứa các con trỏ của các bảng của sort, quicksort và partition. Con trỏ của bảng hiện hành nằm trên đỉnh Stack.

**offset** là một Stack khác để lưu trữ offset. Chú ý rằng với lược đồ  $A \rightarrow BC \{action\ A\}$  thì tất cả các hành vi của các cây con B và C được thực hiện trước A.

Do đó hành vi kết hợp với M được thực hiện trước. Nó không tạo ra bảng ký hiệu cho tầng ngoài cùng (chương trình sort) bằng cách dùng **mktable(nil)**, con trỏ tới bảng này được đưa vào trong Stack tblptr đồng thời được đưa vào Stack offset.

Ký hiệu chưa kết thúc N đóng vai trò tương tự như M khi một khai báo chương trình con xuất hiện. Nó dùng **mktable(top(tblptr))** để tạo ra một bảng mới. Tham số **top(tblptr)** cho giá trị con trỏ tới bảng lại được push vào đỉnh Stack tblptr và 0 được push vào Stack offset.

Với mỗi khai báo biến id:T; một ô mới được tạo ra trong bảng ký hiệu hiện hành. Giá trị trong top(offset) được tăng lên bởi T.width.

Khi hành vi về phải  $P \rightarrow proc\ id ; N D_1 ; S$  diễn ra, kích thước của tất cả các đối tượng dữ liệu khai báo trong D1 sẽ nằm trên đỉnh Stack offset. Nó được lưu trữ bằng cách dùng addwidth, các Stack tblptr và offset bị pop và chúng ta trở về để thao tác trên các khai báo của chương trình con

### 3. Xử lý đối với mẫu tin

Khai báo một mẫu tin được cho bởi luật sinh

$T \rightarrow \text{record } D \text{ end}$

Luật dịch tương ứng

$T \rightarrow \text{record } L D \text{ end}$	$\{ T.type := record(top(tblptr));$ $T.width := top(offset); pop(tblptr) ; pop(offset) \}$
$L \rightarrow \varepsilon$	$\{ t:= mktable(nil); push(t,tblptr) ; push(0,offset) \}$

**Hình 8.13** - Cài đặt bảng ký hiệu cho các tên trường trong mẫu tin

Sau khi từ khóa record được tìm thấy thì hành vi kết hợp với L tạo một bảng ký hiệu mới cho các tên trường. Các hành vi của  $D \rightarrow id : T$  đưa thông tin về tên trường id vào trong bảng ký hiệu cho mẫu tin.

### III. LỆNH GÁN

#### 1. Tên trong bảng ký hiệu

Xét lược đồ dịch để sinh ra mã lệnh 3 địa chỉ cho lệnh gán:

$S \rightarrow id := E$	$\{ p := \text{lookup}(id.name);$ $\text{if } p \neq \text{nil then emit}(p := 'E.place) \text{ else error } \}$
$E \rightarrow E_1 + E_2$	$\{ E.place := \text{newtemp};$ $\text{emit}(E.place := 'E_1.place '+' E_2.place) \}$
$E \rightarrow E_1 * E_2$	$\{ E.place := \text{newtemp};$ $\text{emit}(E.place := 'E_1.place '*' E_2.place) \}$
$E \rightarrow - E_1$	$\{ E.place := \text{newtemp};$ $\text{emit}(E.place := 'unimus' E_1.place) \}$
$E \rightarrow ( E_1 )$	$\{ E.place := E_1.place \}$
$E \rightarrow id$	$\{ p := \text{lookup}(id.name);$ $\text{if } p \neq \text{nil then } E.place := p \text{ else error } \}$

**Hình 8.14** - Lược đồ dịch sinh mã lệnh ba địa chỉ cho lệnh gán

Hàm lookup tìm trong bảng ký hiệu xem có hay không một tên được cho bởi id.name. Nếu có thì trả về con trỏ của ô, nếu không trả về nil.

Xét luật sinh  $D \rightarrow \text{proc id ; ND1 ; S}$

Như trên đã nói, hành vi kết hợp với ký hiệu chưa kết thúc N cho phép con trỏ của bảng ký hiệu cho chương trình con đang nằm trên đỉnh Stack tblptr.

Các tên trong lệnh gán sinh ra bởi ký hiệu chưa kết thúc S sẽ được khai báo trong chương trình con này hoặc trong bao của nó. Khi tham khảo tới một tên thì trước hết hàm lookup sẽ tìm xem có tên đó trong bảng ký hiệu hiện hành hay không. (Bảng danh biểu hiện hành được trỏ bởi top(tblptr)). Nếu không thì dùng con trỏ ở trong header của bảng để tìm bảng ký hiệu bao nó và tìm tên trong đó. Nếu tên không được tìm thấy trong tất cả các mức thì lookup trả về nil.

#### 2. Địa chỉ hóa các phần tử của mảng

Các phần tử của mảng có thể truy xuất nhanh nếu chúng được liên trong một khối các ô nhớ kết 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

Địa chỉ tương đối của  $A[i] = \text{base} + (i - \text{low}) * w$

Trong đó

low: là cận dưới tập chỉ số

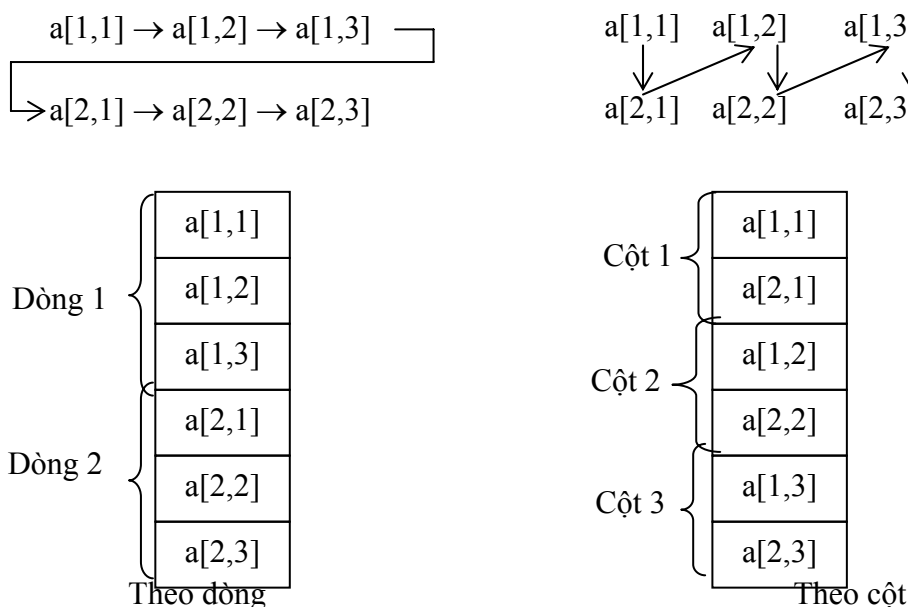
base: là địa chỉ tương đối của ô nhớ cấp phát cho mảng tức là địa chỉ tương đối của  $A[\text{low}]$

Biến đổi một chút ta được

Địa chỉ tương đối của  $A[i] = i * w + (base - low * w)$

Trong đó:  $c = base - low * w$  có thể tính được tại thời gian dịch và lưu trong bảng ký hiệu. Do đó địa chỉ tương đối  $A[i] = i * w + c$ .

Mảng hai chiều có thể xem như là một mảng theo một trong hai dạng: theo dòng (row\_major) hoặc theo cột (column\_major)



**Hình 8.15 - Những cách sắp xếp của mảng hai chiều**

Trong trường hợp lưu trữ theo dòng, địa chỉ tương đối của phần tử  $a[i_1, j_2]$  có thể tính theo công thức

Địa chỉ tương đối của  $A[i_1, j_2] = base + ((i_1 - low_1) * n_2 + j_2 - low_2) * w$

Trong đó  $low_1$  và  $low_2$  là cận dưới của hai tập chỉ số.

$n_2$  : là số các phần tử trong một dòng. Nếu gọi  $high_2$  là cận trên của tập chỉ số thứ 2 thì  $n_2 = high_2 - low_2 + 1$

Trong đó công thức trên chỉ có  $i_1, i_2$  là chưa biết tại thời gian dịch. Do đó, nếu biến đổi công thức để được :

Địa chỉ tương đối của  $A[i_1, j_2] = ((i_1 * n_2) + j_2) * w + (base - ((low_1 * n_2) + low_2) * w)$

Trong đó

$C = (base - ((low_1 * n_2) + low_2) * w)$  được tính tại thời gian dịch và ghi vào trong bảng ký hiệu.

Tổng quát hóa cho trường hợp k chiều, ta có

Địa chỉ tương đối của  $A[i_1, i_2, \dots, i_k]$  là

$((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) w + base - ((\dots((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) w$

### 3. Biến đổi kiểu trong lệnh gán

Giả sử chúng ta có 2 kiểu là integer và real; integer phải đổi thành real khi cần thiết. Ta có, các hành vi ngữ nghĩa kết hợp với luật sinh  $E \rightarrow E_1 + E_2$  như sau:

$E.place := newtemp$

```

if  $E_1.type = integer$  and  $E_2.type = integer$  then begin
    emit( $E.place := 'E_1.place\ int + ' E_2.place$ );
     $E.type := integer$ ;
end
else if  $E_1.type = real$  and  $E_2.type = real$  then begin
    emit( $E.place := 'E_1.place\ real + ' E_2.place$ );
     $E.type := real$ ;
end
else if  $E_1.type = integer$  and  $E_2.type = real$  then begin
     $u := newtemp$ ;    emit( $u := 'intoreal' E_1.place$ );
    emit( $E.place := 'u\ real + ' E_2.place$ );
     $E.type := real$ ;
end
else if  $E_1.type = real$  and  $E_2.type = integer$  then begin
     $u := newtemp$ ;    emit( $u := 'intoreal' E_2.place$ );
    emit( $E.place := 'E_1.place\ real + ' u$ );
     $E.type := real$ ;
end
else  $E.type := type\_error$ ;
end

```

**Hình 8.16** - Hành vi ngữ nghĩa của  $E \rightarrow E_1 + E_2$

**Ví dụ 8.5:** Với lệnh gán  $x := y + i * j$  trong đó  $x, y$  được khai báo là *real*;  $i, j$  được khai báo là *integer*. Mã lệnh 3 địa chỉ xuất ra là:

```

t1 := i int * j
t3 := intoreal t1
t2 := y real + t3
x := t2

```

#### IV. BIỂU THỨC LOGIC

Biểu thức logic được sinh ra bởi văn phạm sau:

$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid id \text{ relop } id \mid \text{true} \mid \text{false}$

Trong đó *or* và *and* kết hợp trái; *or* có độ ưu tiên thấp nhất, kế tiếp là *and* và sau cùng là *not*

Thông thường có 2 phương pháp chính để biểu diễn giá trị logic.

**Phương pháp 1:** Mã hóa *true* và *false* bằng các số và việc đánh giá biểu thức được thực hiện tương tự như đối với biểu thức số học, có thể biểu diễn *true* bởi 1, *false* bởi 0; hoặc các số khác không biểu diễn *true*, số không biểu diễn *false*...

**Phương pháp 2:** Biểu diễn giá trị của biểu thức logic bởi một vị trí đến trong chương trình. Phương pháp này rất thuận lợi để cài đặt biểu thức logic trong các điều khiển.

### 1. Biểu diễn số

Sử dụng 1 để biểu diễn true và 0 để biểu diễn false. Biểu thức được đánh giá từ trái sang phải theo cách tương tự biểu thức số học.

**Ví dụ 8.6:** Với biểu thức **a or b and not c**, ta có dãy lệnh 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`. dãy lệnh 3 địa chỉ tương ứng là

```
100 : if a < b goto 103
101 : t := 0
102 : goto 104
103 : t := 1
104 :
```

Ta có, lược đồ dịch để sinh ra mã lệnh 3 địa chỉ đối với biểu thức logic:

```
E → E1 or E2    { E.place := newtemp; emit(E.place := ' E1.place 'or' E2.place) }
E → E1 and E2    { E.place := newtemp; emit(E.place := ' E1.place 'and' E2.place) }
E → not E1       { E.place := newtemp; emit(E.place := 'not' E1.place ) }
E → id1 relop id2 { E.place := newtemp;
                    emit('if' id1.place relop.op id2.place 'goto' nextstat + 3);
                    emit(E.place := '0'); emit('goto' nextstat + 2);
                    emit(E.place := '1') }
E → true           { E.place := newtemp; emit(E.place := '1') }
E → false          { E.place := newtemp; emit(E.place := '0') }
```

**Hình 8.17** - Lược đồ dịch sử dụng biểu diễn số để sinh mã lệnh ba địa chỉ cho các biểu thức logic

**Ví dụ 8.7:** Với biểu thức **a < b or c < d and e < f**, nó sẽ sinh ra lệnh địa chỉ như sau:

```
100 : if a < b goto 103
101 : t1 := 0
102 : goto 104
103 : t1 := 1
104 : if c < d goto 107
105 : t2 := 0
106 : goto 108
107 : t2 := 1
```

```

108 : if e<f goto 111
109 : t3 := 0
110 : goto 112
111 : t3 := 1
112 : t4 := t2 and t3
113 : t5 := t1 or t4

```

**Hình 8.18** - Sự biên dịch sang mã lệnh ba địa chỉ cho  $a < b$  or  $c < d$  and  $e < f$

## 1. Mã nhảy

Đánh giá biểu thức logic mà không sinh ra mã lệnh cho các toán tử or, and và not. Chúng ta chỉ biểu diễn giá trị một biểu thức bởi vị trí trong chuỗi mã. Ví dụ, trong chuỗi mã lệnh trên, giá trị t1 sẽ phụ thuộc vào việc chúng ta chọn lệnh 101 hay lệnh 103. Do đó giá trị của t1 là thừa.

## 2. Các lệnh điều khiển

```

S →   if E then S1
      | if E then S1 else S2
      | while E do S1

```

Với mỗi biểu thức logic E, chúng ta kết hợp với 2 nhãn

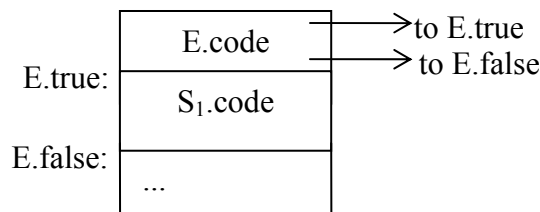
E.true : Nhãn của dòng điều khiển nếu E là true.

E.false : Nhãn của dòng điều khiển nếu E là false.

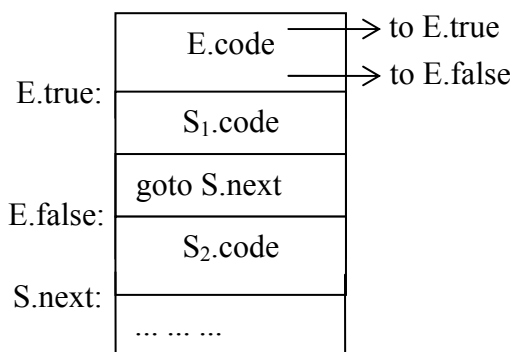
S.code : Mã lệnh 3 địa chỉ được sinh ra bởi S.

S.next : Là nhãn mà lệnh 3 địa chỉ đầu tiên sẽ thực hiện sau mã lệnh của S.

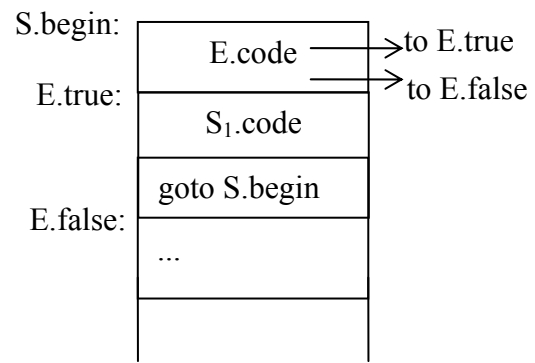
S.begin : Nhãn chỉ định lệnh đầu tiên được sinh ra cho S.



(a) if-then



(b) if-then-else



(c) while-do

**Hình 8.19** - Mã lệnh của các lệnh if-then, if-then-else, và while-do

Ta có định nghĩa trực tiếp cú pháp cho các lệnh điều khiển

Luật sinh	Luật ngữ nghĩa
$S \rightarrow \text{if } E \text{ then } S_1$	$E.true := \text{newlabel};$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.true := \text{newlabel};$ $E.false := \text{newlabel};$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code \parallel$ $\text{gen}(\text{'goto' } S.next) \parallel$ $\text{gen}(E.false ':') \parallel S_2.code$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.begin := \text{newlabel};$ $E.true := \text{newlabel};$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := \text{gen}(S.begin ':') \parallel E.code \parallel \text{gen}(E.true ':') \parallel$ $S_1.code \parallel \text{gen}(\text{'goto' } S.begin)$

**Hình 8.20** - Định nghĩa trực tiếp cú pháp của dòng điều khiển

### 3. Dịch biểu thức logic trong các lệnh điều khiển

- Nếu E có dạng **a<b** thì mã lệnh sinh ra có dạng  
 $\text{if } a < b \text{ goto } E.true$   
 $\text{goto } E.false$
- Nếu E có dạng **E<sub>1</sub> or E<sub>2</sub>**. Nếu E<sub>1</sub> là true thì E là true. Nếu E<sub>1</sub> là false thì phải đánh giá E<sub>2</sub>. Do đó E<sub>1</sub>.false là nhãn của lệnh đầu tiên của E<sub>2</sub>. E sẽ true hay false phụ thuộc vào E<sub>2</sub> là true hay false.
- Tương tự cho **E<sub>1</sub> and E<sub>2</sub>**.
- Nếu E có dạng **not E<sub>1</sub>** thì E<sub>1</sub> là true thì E là false và ngược lại.

Ta có định nghĩa trực tiếp cú pháp cho việc dịch các biểu thức logic thành mã lệnh 3 địa chỉ. Chú ý true và false là các thuộc tính kế thừa.

Luật sinh	Luật ngữ nghĩa
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := \text{newlabel};$ $E_2.true := E.true;$

$E \rightarrow E_1 \text{ and } E_2$	$E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E.false ':') \parallel E_2.code$ $E_1.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E.true ':') \parallel E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow id_1 \text{ relop } id_2$	$E.code := gen('if id_1.place \text{ relop } id_2.place$ $\quad \quad \quad 'goto' E.true) \parallel gen('goto' E.false)$
$E \rightarrow \text{true}$	$E.code := gen('goto' E.true)$
$E \rightarrow \text{false}$	$E.code := gen('goto' E.false)$

**Hình 8.21** - Định nghĩa trực tiếp cú pháp sinh mã lệnh ba địa chỉ cho biểu thức logic

#### 4. Biểu thức logic và biểu thức số học

Trong thực tế biểu thức logic thường chứa những biểu thức số học như  $(a+b) < c$ . Trong các ngôn ngữ mà false có giá trị số là 0 và true có giá trị số là 1 thì  $(a < b) + (b < a)$  có thể được xem như là một biểu thức số học có giá trị 0 nếu  $a = b$  và có giá trị 1 nếu  $a \neq b$ .

Phương pháp biểu diễn biểu thức logic bằng mã lệnh nhảy có thể vẫn còn được sử dụng.

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/và 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.

Ta có luật ngữ nghĩa kết hợp với  $E \rightarrow E_1 + E_2$  như sau

$E.type := arith;$

**if**  $E_1.type = arith$  **and**  $E_2.type = arith$  **then begin**

*/\* phép cộng số học bình thường \*/*

$E.place := newtemp;$

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

**end**

**else if**  $E_1.type = arith$  **and**  $E_2.type = bool$  **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 ...

```

**Hình 8.22** - Luật ngữ nghĩa cho luật sinh  $E \rightarrow E1 + E2$

Trong trường hợp nếu có biểu thức logic nào có biểu thức số học, chúng ta sinh mã lệnh cho E1, E2 bởi các lệnh

```

E2.true : E.place := E1.place +1
        goto nextstat +1
E2.false : E.place := E1.place

```

## V. LỆNH CASE

Lệnh CASE hoặc SWITCH thường được sử dụng trong các ngôn ngữ lập trình.

### 1. Cú pháp của lệnh SWITCH/ CASE

```

SWITCH E
begin
    case V1 : S1
    case V2 : S2
    ....
    case Vn-1 : Sn-1
    default:    Sn
end

```

**Hình 8.23** - Cú pháp của câu lệnh switch

### 2. Dịch trực tiếp cú pháp lệnh Case

1. Đánh giá biểu thức.
2. Tùy một giá trị trong danh sách các case bằng giá trị của biểu thức. Nếu không tìm thấy thì giá trị default của biểu thức được xác định.
3. Thực hiện các lệnh kết hợp với giá trị tìm được để cài đặt.

**Ta có phương pháp cài đặt như sau**

```

    mã lệnh để đánh giá biểu thức E vào t
    goto test
L1 :    mã lệnh của S1
    goto next
L2:    mã lệnh của S2

```

```

goto next
.....
Ln-1 :   mã lệnh của Sn-1
        goto next
Ln :     mã lệnh của Sn
        goto next
test :   if t=V1 goto L1
        if t=V2 goto L2
        .. . . .
        if t=Vn-1 goto Ln-1
        else goto Ln
next:

```

**Hình 8.24** - Dịch câu lệnh case

**Một phương pháp khác để cài đặt lệnh SWITCH là**

```

mã lệnh để đánh giá biểu thức E vào t
if t <> V1 goto L1
mã lệnh của S1
goto next
L1 :   if t <> V2 goto L2
        mã lệnh của S2
        goto next
L2:.....
Ln-2 :   if t <> Vn-1 goto Ln-1
        mã lệnh của Sn-1
        goto next
Ln-1 :   mã lệnh của Sn
next:

```

**Hình 8.24** - Một cách dịch khác của câu lệnh case

## BÀI TẬP CHƯƠNG VIII

**8.1.** Dịch biểu thức :  $a * - (b + c)$  thành các dạng:

- a) Cây cú pháp.
- b) Ký pháp hậu tố.
- c) Mã lệnh máy 3 - địa chỉ.

**8.2.** Trình bày cấu trúc lưu trữ biểu thức  $-(a + b) * (c + d) + (a + b + c)$  ở các dạng:

- a) Bộ tứ .
- b) Bộ tam.
- c) Bộ tam gián tiếp.

**8.3.** Sinh mã trung gian ( dạng mã máy 3 - địa chỉ) cho các biểu thức C đơn giản sau:

- a)  $x = 1$
- b)  $x = y$
- c)  $x = x + 1$
- d)  $x = a + b * c$
- e)  $x = a / (b + c) - d * (e + f)$

**8.4.** Sinh mã trung gian ( dạng mã máy 3 - địa chỉ) cho các biểu thức C sau:

- a)  $x = a[i] + 11$
- b)  $a[i] = b[c[j]]$
- c)  $a[i][j] = b[i][k] * c[k][j]$
- d)  $a[i] = a[i] + b[j]$
- e)  $a[i] += b[j]$

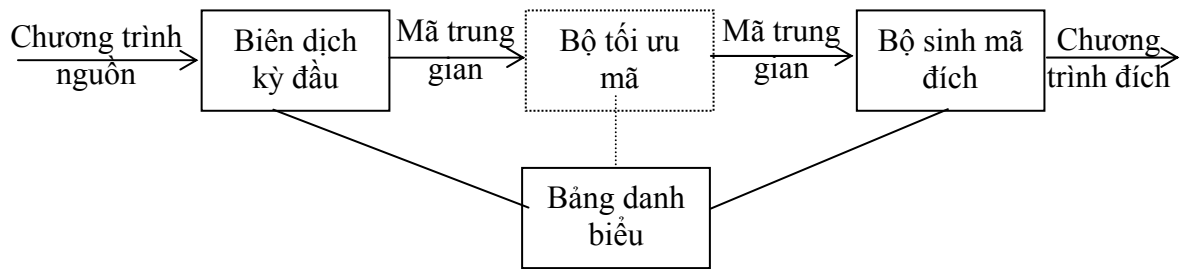
**8.5.** Dịch lệnh gán sau thành mã máy 3 - địa chỉ:

$$A[i, j] := B[i, j] + C[A[k, l]] + D[i + j]$$

## CHƯƠNG IX SINH MÃ ĐÍCH

### Nội dung chính:

Giai đoạn cuối của quá trình biên dịch là *sinh mã đích*. Dữ liệu nhập của bộ sinh mã đích là biểu diễn trung gian của chương trình nguồn và dữ liệu xuất của nó là một chương trình đích (hình 9.1). Kỹ thuật sinh mã đích được trình bày trong chương này không phụ thuộc vào việc dùng hay không dùng giai đoạn tối ưu mã trung gian.



**Hình 9.1-** Vị trí của bộ sinh mã đích

Nhìn chung một *bộ sinh mã đích* phải đảm bảo chạy hiệu quả và phải tạo ra chương trình đích đúng sử dụng hiệu quả tài nguyên của máy đích. Về mặt lý thuyết, vấn đề sinh mã tối ưu là không thực hiện được. Trong thực tế, ta có thể chọn những kỹ thuật heuristic để tạo ra mã tốt nhưng không nhất thiết là mã tối ưu. Chương này đề cập đến *các vấn đề cần quan tâm khi thiết kế một bộ sinh mã*. Bên cạnh đó một *bộ sinh mã đích đơn giản* từ chuỗi các lệnh ba địa chỉ cũng được giới thiệu.

### Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải:

- Nắm được các vấn đề cần chú ý khi thiết kế bộ sinh mã đích.
- Biết cách tạo ra một bộ sinh mã đích đơn giản từ chuỗi các mã lệnh ba địa chỉ. Từ đó có thể mở rộng bộ sinh mã này cho phù hợp với ngôn ngữ lập trình cụ thể.

### Kiến thức cơ bản:

Sinh viên phải có kiến thức về kiến trúc máy tính đặc biệt là phần hợp ngữ (assembly language) để thuận tiện cho việc tiếp nhận kiến thức về máy đích.

### Tài liệu tham khảo:

- [1] **Compilers : Principles, Technique and Tools** - Alfred V.Aho, Jeffrey D.Ullman - Addison - Wesley Publishing Company, 1986.
- [2] **Design of Compilers : Techniques of Programming Language Translation** - Karen A. Lemone - CRC Press, Inc, 1992.

# I. CÁC VẤN ĐỀ THIẾT KẾ BỘ SINH MÃ

Trong khi thiết kế bộ sinh mã, các vấn đề chi tiết như quản trị bộ nhớ, chọn chỉ thị cấp phát thanh ghi và đánh giá thứ tự thực hiện ... phụ thuộc rất nhiều vào ngôn ngữ đích và hệ điều hành.

## 1. Dữ liệu vào của bộ sinh mã

Dữ liệu vào của bộ sinh mã gồm biểu diễn trung gian của chương trình nguồn, cùng thông tin trong bảng danh biểu được dùng để xác định địa chỉ của các đối tượng dữ liệu trong thời gian thực thi. Các đối tượng dữ liệu này được tượng trưng bằng tên trong biểu diễn trung gian. Biểu diễn trung gian của chương trình nguồn có thể ở một trong các dạng: Ký pháp hậu tố, mã ba địa chỉ, cây cú pháp, DAG.

## 2. Dữ liệu xuất của bộ sinh mã – Chương trình đích

Giống như mã trung gian, dữ liệu xuất của bộ sinh mã có thể ở một trong các dạng: Mã máy tuyệt đối, mã máy khả định vị địa chỉ hoặc hợp ngữ.

Việc tạo ra một chương trình đích ở dạng mã máy tuyệt đối cho phép chương trình này được lưu vào bộ nhớ và được thực hiện ngay.

Nếu chương trình đích ở dạng mã máy khả định vị địa chỉ (module đối tượng) thì hệ thống cho phép các chương trình con được biên dịch riêng rẽ. Một tập các module đối tượng có thể được liên kết và tải vào bộ nhớ để thực hiện nhờ bộ tải liên kết (linking loader). Mặc dù ta phải trả giá về thời gian cho việc liên kết và tải vào bộ nhớ các module đã liên kết nếu ta tạo ra các module đối tượng khả định vị địa chỉ. Nhưng bù lại, ta có sự mềm dẻo về việc biên dịch các chương trình con riêng rẽ và có thể gọi một chương trình con đã được biên dịch trước đó từ một module đối tượng. Nếu mã đích không tự động tái định vị địa chỉ, trình biên dịch phải cung cấp thông tin về tái định vị cho bộ tải (loader) để liên kết các chương trình đã được biên dịch lại với nhau.

Việc tạo ra chương trình đích ở dạng hợp ngữ cho phép ta dùng bộ biên dịch hợp ngữ để tạo ra mã máy.

## 3. Lựa chọn chỉ thị

Tập các chỉ thị của máy đích sẽ xác định tính phức tạp của việc lựa chọn chỉ thị. Tính chuẩn và hoàn chỉnh của tập chỉ thị là những yếu tố quan trọng. Nếu máy đích không cung cấp một mẫu chung cho mỗi kiểu dữ liệu thì mỗi trường hợp ngoại lệ phải xử lý riêng. Tốc độ chỉ thị và sự biểu diễn của máy cũng là những yếu tố quan trọng. Nếu ta không quan tâm đến tính hiệu quả của chương trình đích thì việc lựa chọn chỉ thị sẽ đơn giản hơn. Với mỗi lệnh ba địa chỉ ta có thể phác họa một bộ khung cho mã đích. Giả sử lệnh ba địa chỉ dạng  $x := y + z$ , với  $x, y, z$  được cấp phát tĩnh, có thể được dịch sang chuỗi mã đích:

*MOV y, R0 /\* Lưu y vào thanh ghi R0 \*/*

*ADD z, R0 /\* cộng z vào nội dung R0, kết quả chứa trong R0 \*/*

*MOV R0, x /\* lưu nội dung R0 vào x \*/*

Tuy nhiên việc sinh mã cho chuỗi các lệnh ba địa chỉ sẽ dẫn đến sự dư thừa mã. Chẳng hạn với:

$$a := b + c$$

$$d := a + e$$

ta chuyển sang mã đích:

*MOV b, R<sub>o</sub>*

*ADD c, R<sub>o</sub>*

*MOV R<sub>o</sub>, a*

*MOV a, R<sub>0</sub>*

*ADD e, R<sub>o</sub>*

*MOV R<sub>o</sub>, d*

và ta nhận thấy rằng chỉ thị thứ tư là thừa.

Chất lượng mã được tạo ra được xác định bằng tốc độ và kích thước của mã. Một máy đích có tập chỉ thị phong phú có thể sẽ cung cấp nhiều cách để hiện thực một tác vụ cho trước. Điều này có thể dẫn đến tốc độ thực hiện chỉ thị rất khác nhau. Chẳng hạn, nếu máy đích có chỉ thị INC thì câu lệnh ba địa chỉ  $a := a + 1$  có thể được cài đặt chỉ bằng câu lệnh *INC a*. Cách này hiệu quả hơn là dùng chuỗi các chỉ thị sau:

*MOV a, R<sub>o</sub>*

*ADD #1, R<sub>o</sub>*

*MOV R<sub>o</sub>, a*

Như ta đã nói, tốc độ của chỉ thị là một trong những yếu tố quan trọng để thiết kế chuỗi mã tốt. Nhưng, thông tin thời gian thường khó xác định.

Việc quyết định chuỗi mã máy nào là tốt nhất cho câu lệnh ba địa chỉ còn phụ thuộc vào ngữ cảnh của nơi chứa câu lệnh đó.

#### 4. Cấp phát thanh ghi

Các chỉ thị dùng toán hạng thanh ghi thường ngắn hơn và nhanh hơn các chỉ thị dùng toán hạng trong bộ nhớ. Vì thế, hiệu quả của thanh ghi đặc biệt quan trọng trong việc sinh mã tốt. Ta thường dùng thanh ghi trong hai trường hợp:

1. Trong khi cấp phát thanh ghi, ta lựa chọn tập các biến lưu trữ trong các thanh ghi tại một thời điểm trong chương trình.

2. Trong khi gán thanh ghi, ta lấy ra thanh ghi đặc biệt mà biến sẽ thường trú trong đó.

Việc tìm kiếm một lệnh gán tối ưu của thanh ghi, ngay với cả các giá trị thanh ghi đơn, cho các biến là một công việc khó khăn. Vấn đề càng trở nên phức tạp hơn vì phần cứng và / hoặc hệ điều hành của máy đích yêu cầu qui ước sử dụng thanh ghi.

##### 1. Lựa chọn cho việc đánh giá thứ tự

Thứ tự thực hiện tính toán có thể ảnh hưởng đến tính hiệu quả của mã đích. Một số thứ tự tính toán có thể cần ít thanh ghi để lưu giữ các kết quả trung gian hơn các thứ tự tính toán khác. Việc lựa chọn được thứ tự tốt nhất là một vấn đề khó. Ta nên tránh vấn đề này bằng cách sinh mã cho các lệnh ba địa chỉ theo thứ tự mà chúng đã được sinh ra bởi bộ mã trung gian.

##### 2. Sinh mã

Tiêu chuẩn quan trọng nhất của bộ sinh mã là phải tạo ra mã đúng. Tính đúng của mã có một ý nghĩa rất quan trọng. Với những quy định về tính đúng của mã, việc thiết kế bộ sinh mã sao cho nó được thực hiện, kiểm tra, bảo trì đơn giản là mục tiêu thiết kế quan trọng.

## II. MÁY ĐÍCH

Trong chương trình này, chúng ta sẽ dùng máy đích như là **máy thanh ghi** (register machine). Máy này tượng trưng cho máy tính loại trung bình. Tuy nhiên, các kỹ thuật sinh mã được trình bày trong chương này có thể dùng cho nhiều loại máy tính khác nhau.

Máy đích của chúng ta là máy tính địa chỉ byte với mỗi từ gồm bốn byte và có  $n$  thanh ghi :  $R_0, R_1 \dots R_{n-1}$ . Máy đích gồm các chỉ thị hai địa chỉ có dạng chung:

### **op source, destination**

Trong đó **op** là mã tác vụ. **Source** (nguồn) và **destination** (đích) là các trường dữ liệu. Ví dụ một số mã tác vụ:

*MOV chuyển source đến destination*

*ADD cộng source và destination*

*SUB trừ source cho destination*

Source và destination của một chỉ thị được xác định bằng cách kết hợp các thanh ghi và các vị trí nhớ với các mode địa chỉ. Mô tả content (a) biểu diễn cho nội dung của thanh ghi hoặc địa chỉ của bộ nhớ được biểu diễn bởi a.

### **mode địa chỉ cùng với dạng hợp ngữ và giá kết hợp:**

Mode	Dạng	Địa chỉ	Giá
<i>Absolute</i>	<i>M</i>	<i>M</i>	<i>1</i>
<i>Register</i>	<i>R</i>	<i>R</i>	<i>0</i>
<i>Indexed</i>	<i>c(R)</i>	<i>c + contents ( R )</i>	<i>1</i>
<i>Indirect register</i>	<i>*R</i>	<i>contents ( R )</i>	<i>0</i>
<i>Indirect indexed</i>	<i>*c(R)</i>	<i>contents ( c + contents ( R ) )</i>	<i>1</i>

Vị trí nhớ  $M$  hoặc thanh ghi  $R$  biểu diễn chính nó khi được sử dụng như một nguồn hay đích. Độ dời địa chỉ  $c$  từ giá trị trong thanh ghi  $R$  được viết là  $c(R)$ .

### **Chẳng hạn:**

1. *MOV R0, M* : Lưu nội dung của thanh ghi  $R_0$  vào vị trí nhớ  $M$ .
2. *MOV 4(R0), M* : Xác định một địa chỉ mới bằng cách lấy độ dời tương đối (offset) 4 cộng với nội dung của  $R_0$ , sau đó lấy nội dung tại địa chỉ này,  $contains(4 + contains(R_0))$ , lưu vào vị trí nhớ  $M$ .
3. *MOV \*4(R0), M* : Lưu giá trị  $contents(contains(4 + contents(R_0)))$  vào vị trí nhớ  $M$ .
4. *MOV #1, R0* : Lấy hằng 1 lưu vào thanh ghi  $R_0$ .

### **Giá của chỉ thị**

Giá của chỉ thị (instruction cost) được tính bằng một cộng với giá kết hợp mode địa chỉ nguồn và đích trong bảng trên. Giá này tượng trưng cho chiều dài của chỉ thị. Mode địa chỉ dùng thanh ghi sẽ có giá bằng không và có giá bằng một khi nó dùng vị trí nhớ hoặc hằng. Nếu vấn đề vị trí nhớ là quan trọng thì chúng ta nên tối thiểu hóa chiều dài chỉ thị. Đối với phần lớn các máy và phần lớn các chỉ thị, thời gian cần để lấy một chỉ thị từ bộ nhớ bao

giờ cũng xảy ra trước thời gian thực hiện chỉ thị. Vì vậy, bằng việc tối thiểu hóa độ dài chỉ thị, ta còn tối thiểu hoá được thời gian cần để thực hiện chỉ thị.

*Một số minh họa việc tính giá của chỉ thị:*

1. Chỉ thị MOV R0, R1 : Sao chép nội dung thanh ghi R0 vào thanh ghi R1. Chỉ thị này có giá là một vì nó chỉ chiếm một từ trong bộ nhớ .

2. MOV R5, M: Sao chép nội dung thanh ghi R5 vào vị trí nhớ M. Chỉ thị này có giá trị là hai vì địa chỉ của vị trí nhớ M là một từ sau chỉ thị.

3. Chỉ thị ADD #1, R3: cộng hằng 1 vào nội dung thanh ghi R3. Chỉ thị có giá là hai vì hằng 1 phải xuất hiện trong từ kế tiếp sau chỉ thị.

4. Chỉ thị SUB 4(R0), \*12(R1) : Lưu giá trị của contents (contents (12 + contents (R1))) - contents (4 + contents (R0)) vào đích \*12(R1). Giá của chỉ thị này là ba vì hằng 4 và 12 được lưu trữ trong hai từ kế tiếp theo sau chỉ thị.

Với mỗi câu lệnh ba địa chỉ, ta có thể có nhiều cách cài đặt khác nhau. Ví dụ câu lệnh  $a := b + c$  - trong đó b và c là biến đơn, được lưu trong các vị trí nhớ phân biệt có tên b, c - có những cách cài đặt sau:

1.  $MOV\ b, R_0$   
 $ADD\ c, R_0$  *giá = 6*  
 $MOV\ R_0, a$
2.  $MOV\ b, a$  *giá = 6*  
 $ADD\ c, a$

3. Giả sử thanh ghi R0, R1, R2 giữ địa chỉ của a, b, c. Chúng ta có thể dùng hai địa chỉ sau cho việc sinh mã lệnh:

- $$a := b + c \Rightarrow$$
- $$MOV\ *R_1, *R_0 \quad \text{giá} = 2$$
- $$ADD\ *R_2, *R_0$$

4. Giả sử thanh ghi R1 và R2 chứa giá trị của b và c và trị của b không cần lưu lại sau lệnh gán. Chúng ta có thể dùng hai chỉ thị sau:

- $$ADD\ R_2, R_1 \quad \text{giá} = 3$$
- $$MOV\ R_1, a$$

Như vậy, với mỗi cách cài đặt khác nhau ta có những giá khác nhau. Ta cũng thấy rằng muốn sinh mã tốt thì phải hạ giá của các chỉ thị . Tuy nhiên việc làm khó mà thực hiện được. Nếu có những quy ước trước cho thanh ghi, lưu giữ địa chỉ của vị trí nhớ chứa giá trị tính toán hay địa chỉ để đưa trị vào, thì việc lựa chọn chỉ thị sẽ dễ dàng hơn.

### III. QUẢN LÝ BỘ NHỚ TRONG THỜI GIAN THỰC HIỆN

Trong phần này ta sẽ nói về việc sinh mã để quản lý các mẫu tin hoạt động trong thời gian thực hiện. Hai chiến lược cấp phát bộ nhớ chuẩn được trình bày trong chương VII là cấp phát tĩnh và cấp phát Stack. Với cấp phát tĩnh, vị trí của mẫu tin hoạt động trong bộ nhớ được xác định trong thời gian biên dịch. Với cấp phát Stack, một mẫu tin hoạt động được đưa vào Stack khi có sự thực hiện một thủ tục và được lấy ra khỏi Stack khi hoạt động kết thúc. Ở đây, ta sẽ xem xét cách thức mã đích của một thủ tục tham chiếu tới các đối tượng dữ liệu trong

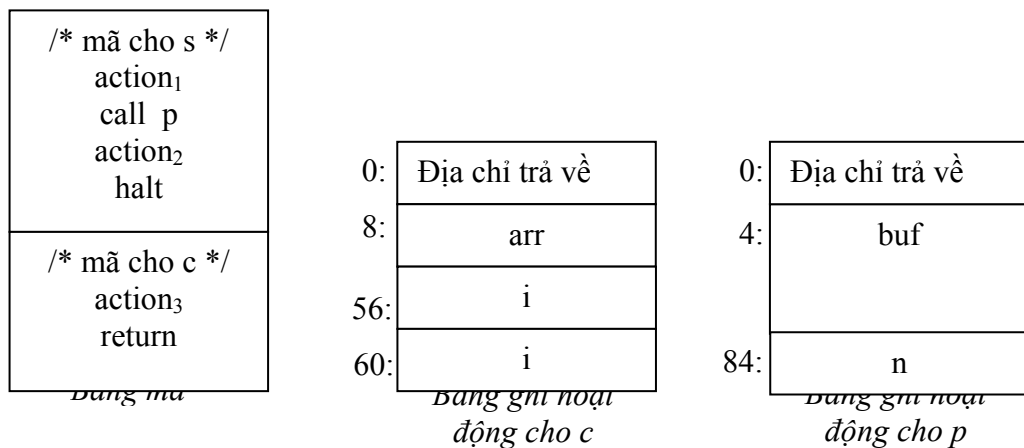


các mẫu tin hoạt động. Như ta đã nói ở chương VII, một mẫu tin hoạt động cho một thủ tục có các trường: tham số, kết quả, thông tin về trạng thái máy, dữ liệu cục bộ, lưu trữ tạm thời và cục bộ, và các liên kết. Trong phần này, ta minh họa các chiến lược cấp phát sử dụng trường trạng thái để giữ giá trị trả về và dữ liệu cục bộ, các trường còn lại được dùng như đã đề cập ở chương VII.

Việc cấp phát và giải phóng các mẫu tin hoạt động là một phần trong chuỗi hành vi gọi và trả về của chương trình con. Ta quan tâm đến việc sinh mã cho các lệnh sau:

1. *call*
2. *return*
3. *halt*
4. *action* /\* tượng trưng cho các lệnh khác \*/

Chẳng hạn, mã ba địa chỉ, chỉ chứa các loại câu lệnh trên, cho các chương trình c và p cũng như các mẫu tin hoạt động của chúng:



**Hình 9.2** – Dữ liệu vào của bộ sinh mã

Kích thước và việc xếp đặt các mẫu tin được kết hợp với bộ sinh mã nhờ thông tin về tên trong bảng danh biểu.

Ta giả sử bộ nhớ thời gian thực hiện được phân chia thành các vùng cho mã, dữ liệu tĩnh và Stack.

## 1. Cấp phát tĩnh

Chúng ta sẽ xét các chỉ thị cần thiết để thực hiện việc cấp phát tĩnh. Lệnh *call* trong mã trung gian được thực hiện bằng dãy hai chỉ thị đích. Chỉ thị *MOV* lưu địa chỉ trả về. Chỉ thị *GOTO* chuyển quyền điều khiển cho chương trình được gọi.

*MOV* #here + 20, callee.static\_area

*GOTO* callee.code\_area

Các thuộc tính *callee.static\_area* và *callee.code\_area* là các hằng tham chiếu tới các địa chỉ của mẫu tin hoạt động và chỉ thị đầu tiên trong đoạn mã của chương trình con được gọi. #here + 20 trong chỉ thị *MOV* là địa chỉ trả về. Nó cũng chính là địa chỉ của chỉ thị đứng sau lệnh *GOTO*. Mã của chương trình con kết thúc bằng lệnh trả về chương trình gọi, trừ chương trình chính, đó là lệnh *halt*. Lệnh này trả quyền điều khiển cho hệ điều hành. Lệnh trả về được dịch sang mã máy là *GOTO \*callee\_static\_area* thực hiện việc chuyển quyền điều khiển về địa chỉ được lưu giữ ở ô nhớ đầu tiên của mẫu tin hoạt động.

**Ví dụ 9.1:** Mã đích trong chương trình sau được tạo ra từ các chương trình con c và p ở hình 9.2. Giả sử rằng: các mã đó được lưu tại địa chỉ bắt đầu là 100 và 200, mỗi chỉ thị action chiếm 20 byte, và các mẫu tin hoạt động cho c và p được cấp phát tĩnh bắt đầu tại các địa chỉ 300 và 364. Ta dùng chỉ thị action để thực hiện câu lệnh action. Như vậy, mã đích cho các chương trình con:

```

/* mã cho c */
100: ACTION1
120: MOV #140, 364/* lưu địa chỉ trả về 140 */
132: GOTO 200 /* gọi p */
140: ACTION2
160: HALT

/* mã cho p */
200: ACTION3
220: GOTO *364 /* trả về địa chỉ được lưu tại vị trí 364 */
/* 300-364 lưu mẫu tin hoạt động của c */
300: /* chứa địa chỉ trả về */
304: /* dữ liệu cục bộ của c */
/* 364 - 451 chứa mẫu tin hoạt động của p */
364: /* chứa địa chỉ trả về */
368: /* dữ liệu cục bộ của p */

```

**Hình 9.3** - Mã đích cho dữ liệu vào của hình 9.2

Sự thực hiện bắt đầu bằng chỉ thị action tại địa chỉ 100. Chỉ thị MOV ở địa chỉ 120 sẽ lưu địa chỉ trả về 140 vào trường trạng thái máy, là từ đầu tiên trong mẫu tin hoạt động của p. Chỉ thị GOTO 200 sẽ chuyển quyền điều khiển về chỉ thị đầu tiên trong đoạn mã của chương trình con p. Chỉ thị GOTO \*364 tại địa chỉ 132 chuyển quyền điều khiển sang chỉ thị đầu tiên trong mã đích của chương trình con được gọi.

Giá trị 140 được lưu vào địa chỉ 364, \*364 biểu diễn giá trị 140 khi lệnh GOTO tại địa chỉ 220 được thực hiện. Vì thế quyền điều khiển trả về địa chỉ 140 và tiếp tục thực hiện chương trình con c.

## 2. Cấp phát theo cơ chế Stack

Cấp phát tĩnh sẽ trở thành cấp phát Stack nếu ta sử dụng địa chỉ tương đối để lưu giữ các mẫu tin hoạt động. Vị trí mẫu tin hoạt động chỉ được xác định trong thời gian thực thi. Trong cấp phát Stack, vị trí này thường được lưu vào thanh ghi. Vì thế các ô nhớ của mẫu tin hoạt động được truy xuất như là độ dời (offset) so với giá trị trong thanh ghi đó.

Thanh ghi SP chứa địa chỉ bắt đầu của mẫu tin hoạt động của chương trình con nằm trên đỉnh Stack. Khi lời gọi của chương trình con xuất hiện, chương trình bị gọi được cấp phát, SP được tăng lên một giá trị bằng kích thước mẫu tin hoạt động của chương trình gọi và chuyển quyền điều khiển cho chương trình con được gọi. Khi quyền điều khiển trả về cho chương trình gọi, SP giảm đi một khoảng bằng kích thước mẫu tin hoạt động của chương trình gọi. Vì thế, mẫu tin của chương trình con được gọi đã được giải phóng.

Mã cho chương trình con đầu tiên có dạng:



**Hình 9.4 - Mã ba địa chỉ minh họa cấp phát sử dụng Stack**

<i>/* mã cho s */</i> action <sub>1</sub> call q action <sub>2</sub> halt
<i>/* mã cho p */</i> action <sub>3</sub> return
<i>/* mã cho q */</i> action <sub>4</sub> call p action <sub>5</sub> call q action <sub>6</sub> call q return

*/\* mã cho s \*/*

100: MOV #600, SP */\* khởi động Stack \*/*

108: ACTION<sub>1</sub>

128: ADD #ssize, SP */\* chuỗi gọi bắt đầu \*/*

136: MOV #152, \*SP */\* lưu địa chỉ trả về \*/*

144: GOTO 300 */\* gọi q \*/*

152: SUB #ssize, SP */\* Lưu giữ SP \*/*

160: ACTION<sub>2</sub>

180: HALT

*/\* mã cho p \*/*

200: ACTION<sub>3</sub>

220: GOTO \*0(SP) */\* trả về chương trình gọi \*/*

*/\* mã cho q \*/*

300: ACTION<sub>4</sub> */\* nhảy có điều kiện về 456 \*/*

320: ADD #qsize, SP

328: MOV #344, \*SP */\* lưu địa chỉ trả về \*/*

336: GOTO 200 */\* gọi p \*/*

344: SUB #qsize, SP

352: ACTION<sub>5</sub>

372: ADD #qsize, SP

380: MOV #396, \*SP */\* lưu địa chỉ trả về \*/*

388: GOTO 300	/* gọi q */
396: SUB #qsize, SP	
404: ACTION <sub>6</sub>	
424: ADD #qsize, SP	
432: MOV #448, *SP	/* lưu địa chỉ trả về */
440: GOTO 300	/* gọi q */
448: SUB #qsize, SP	
456: GOTO *0(SP)	/* trả về chương trình gọi */
600:	/* địa chỉ bắt đầu của Stack trung tâm */

**Hình 9.5 - Mã đích cho chuỗi ba địa chỉ trong hình 9.4**

Ta giả sử rằng action<sub>4</sub> gồm lệnh nhảy có điều kiện tới địa chỉ 456 có lệnh trả về từ q. Ngược lại chương trình đệ quy q có thể gọi chính nó mãi. Trong ví dụ này chúng ta giả sử lần gọi đầu tiên trên q sẽ không trả về chương trình gọi ngay, nhưng những lần sau thì có thể. SP có giá trị lúc đầu là 600, địa chỉ bắt đầu của Stack. SP lưu giữ giá trị 620 chỉ trước khi chuyển quyền điều khiển từ s sang q vì kích thước của mẫu tin hoạt động s là 20. Khi q gọi p, SP sẽ tăng lên 680 khi chỉ thị tại địa chỉ 320 được thực hiện, Sp chuyển sang 620 sau khi chuyển quyền điều khiển cho chương trình con p. Nếu lời gọi đệ quy của q trả về ngay thì giá trị lain nhất của SP trong suốt quá trình thực hiện là 680. Vị trí được cấp phát theo cơ chế Stack có thể lên đến địa chỉ 739 vì mẫu tin hoạt động của q bắt đầu tại 680 và chiếm 60 byte.

### 3. Địa chỉ của các tên trong thời gian thực hiện

Chiến lược cấp phát lưu trữ và xếp đặt dữ liệu cục bộ trong mẫu tin hoạt động của chương trình con xác định cách thức truy xuất vùng nhớ của tên.

Nếu chúng ta dùng cơ chế cấp phát tĩnh với vùng dữ liệu được cấp phát tại địa chỉ static. Với lệnh gán  $x := 0$ , địa chỉ tương đối của x trong bảng danh biểu là 12. Vậy địa chỉ của x trong bộ nhớ là static + 12. Lệnh gán  $x := 0$  được chuyển sang mã ba địa chỉ static[12] := 0. Nếu vùng dữ liệu bắt đầu tại địa chỉ 100, mã đích cho chỉ thị là:

MOV #0,112

Nếu ngôn ngữ dùng cơ chế display để truy xuất tên không cục bộ, giả sử x là tên cục bộ của chương trình con hiện hành và thanh ghi R3 lưu giữ địa chỉ bắt đầu của mẫu tin hoạt động đó thì chúng ta sẽ dịch lệnh  $x := 0$  sang chuỗi mã ba địa chỉ:

$$t_1 := 12 + R_3$$

$$* t_1 := 0$$

Từ đó ta chuyển sang mã đích:

MOV #0, 12(R<sub>3</sub>)

Chú ý rằng, giá trị thanh ghi R3 không được xác định trong thời gian biên dịch.

## IV. KHỐI CƠ BẢN VÀ LƯU ĐỒ

Đồ thị biểu diễn các lệnh ba địa chỉ, được gọi là lưu đồ, giúp ta hiểu các giải thuật sinh mã ngay cả khi đồ thị không được xác định cụ thể bằng giải thuật sinh mã. Các nút của lưu đồ biểu diễn sự tính toán, các cạnh biểu diễn dòng điều khiển.

### 1. Khối cơ bản

Khối cơ bản (basic block) là 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 chuỗi lệnh ba địa chỉ sau tạo nên một khối cơ bản

$$t_1 := a * a$$

$$t_2 := a * b$$

$$t_3 := 2 * t_2$$

$$t_4 := t_1 + t_2$$

$$t_5 := b * b$$

$$t_6 := t_4 + t_5$$

Lệnh ba địa chỉ  $x := y + z$  dùng các giá trị được chứa ở các vị trí nhớ của  $y, z$  để thực hiện phép cộng và xác định địa chỉ của  $x$  để lưu kết quả phép cộng vào. Một tên trong khối cơ bản được gọi là ‘sống’ tại một điểm nào đó nếu giá trị của nó sẽ được sử dụng sau điểm đó trong chương trình hoặc được dùng ở khối cơ bản khác. Giải thuật sau đây phân chia chuỗi các lệnh ba địa chỉ sang các khối cơ bản.

#### □ Giải thuật 9.1: Phân chia các khối cơ bản

**Input:** Các lệnh ba địa chỉ.

**Output:** Danh sách các khối cơ bản với từng chuỗi các lệnh ba địa chỉ cho từng khối.

**Phương pháp:**

1. Xác định tập các lệnh dẫn đầu (leader), các lệnh đầu tiên của các khối cơ bản, ta dùng các quy tắc sau:

- i) Lệnh đầu tiên là lệnh dẫn đầu.
- ii) Bất kỳ lệnh nào là đích nhảy đến của lệnh GOTO có điều kiện hoặc không điều kiện đều là lệnh dẫn đầu.
- iii) Bất kỳ lệnh nào đi sau lệnh GOTO có điều kiện hoặc không điều kiện đều là lệnh dẫn đầu.

2. Với mỗi lệnh dẫn đầu, khối cơ bản gồm có nó và tất cả các lệnh tiếp theo nhưng không gồm một lệnh dẫn đầu nào khác hay là lệnh kết thúc chương trình.

**Ví dụ 9.3:** Đoạn chương trình sau tính tích vector vô hướng của hai vector  $a$  và  $b$  có độ dài 20.

Begin

prod := 0

i := 1

Repeat

prod: = prod + a [i] \* b[i];

```

        i := i + 1
    Until i > 20
End

```

### Hình 9.6 - Chương trình tính tích vector vô hướng

Đoạn chương trình này được dịch sang mã ba địa chỉ như sau:

```

(1)  prod := 0
(2)  i := 1
(3)  t1 := 4 * i
(4)  t2 := a[t1]    /* tính a[i] */
(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)

```

### Hình 9.7 - Mã ba địa chỉ để tính tích vector vô hướng

Lệnh (1) là lệnh dẫn đầu theo quy tắc i, lệnh (3) là lệnh dẫn đầu theo quy tắc ii và lệnh sau lệnh (12) là lệnh dẫn đầu theo quy tắc iii.

Như vậy lệnh (1) và (2) tạo nên khối cơ bản thứ nhất. Lệnh (3) đến (12) tạo nên khối cơ bản thứ hai.

## 2. Sự chuyển đổi giữa các khối

Khối cơ bản tính các biểu thức. Các biểu thức này là giá trị của các tên “sống” khi ra khỏi khối. Hai khối cơ bản tương đương nhau khi chúng tính các biểu thức giống nhau.

Một số chuyển đổi có thể được áp dụng vào một khối cơ bản mà không làm thay đổi các biểu thức được tính toán trong đó. Nhiều phép chuyển đổi rất có ích vì nó cải thiện chất lượng mã đích được sinh ra từ khối cơ bản. Hai phương pháp chuyển đổi cục bộ quan trọng được áp dụng cho các khối cơ bản là chuyển đổi bảo toàn cấu trúc và chuyển đổi đại số.

### Chuyển đổi bảo toàn cấu trúc

Những chuyển đổi bảo toàn cấu trúc trên các khối cơ bản bao gồm:

1. Loại bỏ các biểu thức con chung.
2. Loại bỏ mã chết.
3. Đặt tên lại các biến tạm.
4. Hoán đổi hai lệnh độc lập kề nhau.

Giả sử trong các khối cơ bản không chứa dãy, con trỏ hay lời gọi chương trình con.

1. Loại bỏ các biểu thức con chung

Khối cơ bản sau:

$$a := b + c$$

$$b := a - d$$

$$c := b + c$$

$$d := a - d$$

Câu lệnh thứ hai và thứ tư tính cùng một biểu thức  $b + c - d$ . Vì vậy, khối cơ bản này được chuyển thành khối tương đương sau:

$$a := b + c$$

$$b := a - d$$

$$c := b + c$$

$$d := b$$

### 2. Loại bỏ mã lệnh chết

Giả sử  $x$  không còn được sử dụng nữa. Nếu câu lệnh  $x := y + z$  xuất hiện trong khối cơ bản thì lệnh này sẽ bị loại mà không làm thay đổi giá trị của khối.

### 3. Đặt lại tên cho biến tạm

Giả sử ta có lệnh  $t := b + c$  với  $t$  là biến tạm. Nếu ta viết lại lệnh này thành  $u := b + c$  mà  $u$  là biến tạm mới và thay  $t$  bằng  $u$  ở bất cứ chỗ nào xuất hiện  $t$  thì giá trị của khối cơ bản sẽ không bị thay đổi. Thực tế, ta có thể chuyển một khối cơ bản sang một khối cơ bản tương đương. Và ta gọi khối cơ bản được tạo ra như vậy là dạng chuẩn.

Giả sử chúng ta một khối với hai câu lệnh kế tiếp:

$$t_1 := b + c$$

$$t_2 := x + y$$

Ta có thể hoán đổi hai lệnh này mà không làm thay đổi giá trị của khối nếu và chỉ nếu  $x$  và  $y$  đều không phải  $t_1$  cũng như  $b$  và  $c$  đều không phải là  $t_2$ . Khối cơ bản có dạng chuẩn cho phép tất cả các lệnh có quyền hoán đổi nếu có thể.

## Chuyển đổi đại số

Các biểu thức trong một khối cơ bản có thể được chuyển đổi sang các biểu thức tương đương. Phép chuyển đổi đại số này giúp ta đơn giản hoá các biểu thức hoặc thay thế các biểu thức có giá cao bằng các biểu thức có giá rẻ hơn.

*Chẳng hạn*, câu lệnh  $x := x + 0$  hoặc  $x := x * 1$  có thể được loại bỏ khỏi khối mà không làm thay đổi giá trị của biểu thức. Toán tử lũy thừa trong câu lệnh  $x := y ** 2$  cần một lời gọi hàm để thực hiện. Tuy nhiên, lệnh này vẫn có thể được thay bằng lệnh tương đương có giá rẻ hơn mà không cần lời gọi hàm.

## 3. Lưu đồ

Ta có thể thêm thông tin về dòng điều khiển vào tập các khối cơ bản bằng việc xây dựng các đồ thị trực tiếp được gọi là lưu đồ (flow graph). Các nút của lưu đồ là khối cơ bản. Một nút được gọi là khối đầu nếu nó chứa lệnh đầu tiên của chương trình. Cạnh nối trực tiếp từ khối  $B_1$  đến khối  $B_2$  nếu  $B_2$  là khối đứng ngay sau  $B_1$  trong một chuỗi thực hiện nào đó. Nghĩa là, nếu:

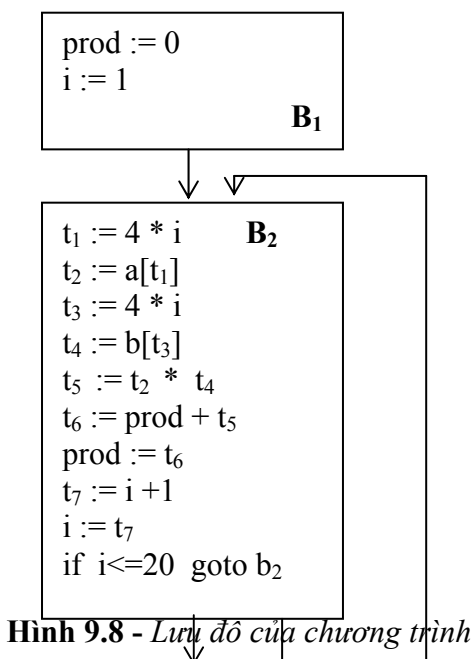
1. Lệnh nhảy không hoặc có điều kiện từ lệnh cuối của  $B_1$  sẽ đi đến lệnh đầu tiên của  $B_2$ .



2.  $B_2$  đứng ngay sau  $B_1$  trong thứ tự của chương trình và  $B_1$  không kết thúc bằng một lệnh nhảy không điều kiện.

Chúng ta nói  $B_1$  là tiền bối (predecessor) của  $B_2$  hay  $B_2$  là hậu duệ (successor) của  $B_1$ .

**Ví dụ 9.4:**



**Hình 9.8** - Lưu đồ của chương trình

#### 4. Biểu diễn các khối cơ bản

Các khối cơ bản được biểu diễn bởi nhiều loại cấu trúc dữ liệu.

Sau khi phân chia các lệnh ba địa chỉ bằng giải thuật 9.1. Mỗi khối cơ bản được biểu diễn bằng một mẫu tin gồm một số bộ tứ, theo sau là một con trỏ trỏ tới lệnh dẫn đầu (bộ tứ đầu tiên) của khối, và một danh sách các tiền bối và hậu duệ của khối.

DAG cũng là một cấu trúc dữ liệu rất thích hợp để thực hiện việc chuyển đổi các khối cơ bản. Xây dựng DAG từ các lệnh ba địa chỉ là một cách tốt để xác định được: Các biểu thức chung (được tính nhiều lần), tên được dùng trong khối nhưng không được dùng khi ra ngoài khối, và các biểu thức mà giá trị của nó được dùng khi ra khỏi khối.

#### □ Giải thuật 9.2: Xây dựng DAG

**Input:** Khối cơ bản

**Output:** DAG cho khối cơ bản, chứa các thông tin sau:

1. Tên cho từng nút. Tên nút lá là danh biểu (hằng số). Tên nút trung gian là toán tử.
2. Với mỗi nút, một danh sách (có thể rỗng) gồm các danh biểu (hằng số không được phép có trong danh sách này).

**Phương pháp:** Giả sử ta đã có cấu trúc dữ liệu để tạo nút có một hoặc hai con. Ta phải phân biệt con bên trái và bên phải đối với những nút có hai con. Ta cũng có vị trí để ghi tên cho mỗi nút và có cơ chế tạo danh sách liên kết của các danh biểu gắn với mỗi nút. Ta cũng giả sử tồn tại hàm node (identifier), khi ta xây dựng DAG, sẽ trả về nút mới nhất có liên quan với identifier. Thực tế node (identifier) là nút biểu diễn giá trị của danh biểu (identifier) tại thời điểm hiện tại trong quá trình xây dựng DAG.

Quá trình xây dựng DAG thực hiện qua các bước từ (1) đến (3) cho mỗi lệnh của khối. Lúc đầu, ta giả sử chưa có các nút và hàm node không được định nghĩa cho tất cả các đối số. Các dạng lệnh ba địa chỉ ở một trong các dạng sau: (i)  $x := y \text{ op } z$ , (ii)  $x := \text{op } y$ , (iii)  $x := y$ .

Trường hợp lệnh điều kiện, chẳng hạn `if i <= 20 goto`, ta coi là trường hợp (i) với `x` không được định nghĩa.

1. Nếu node (`y`) không được định nghĩa, tạo lá có tên `y` và *node* (`y`) chính là nút này. Trong trường hợp (i), nếu node(`z`) không được định nghĩa, ta tạo lá tên `z` và lá chính là node (`z`).

2. Trong trường hợp (i), xác định xem trên DAG có nút nào có tên `op` mà con trái là node (`y`) và con phải là node (`z`). Nếu không thì tạo ra nút có tên `op`, ngược lại `n` là nút đã tìm thấy hoặc đã được tạo. Trong trường hợp (ii), ta xác định xem có nút nào có tên `op`, mà nó chỉ có một con duy nhất là node (`y`). Nếu chưa có nút như trên ta sẽ tạo nút `op` và coi `n` là nút tìm thấy hoặc vừa được tạo ra. Trong trường hợp thứ (iii) thì đặt `n` là *node*(`y`).

3. Xoá `x` ra khỏi danh sách các danh biểu gắn với nút node(`x`). Nối `x` vào danh sách các danh biểu gắn vào nút `n` được tìm ở bước (2) và đặt *node*(`x`) cho `n`.

## 5. Vòng lặp

Vòng lặp (loop) là một tập hợp các nút trong lưu đồ sao cho:

1. Tất cả các nút trong tập hợp phải kết nối chặt chẽ với nhau, nghĩa là phải có con đường với kích thước bằng một hoặc lớn hơn đi từ một nút bất kỳ trong vòng lặp đến một nút bất kỳ khác và nó phải nằm hoàn toàn trong vòng lặp.

2. Các nút lựa chọn này chỉ có duy nhất một lối vào, nghĩa là con đường từ nút ngoài vòng lặp đi vào vòng lặp phải đi qua lối vào đó.

Nếu một vòng lặp không chứa vòng lặp nào khác thì gọi là vòng lặp trong cùng.

## V. THÔNG TIN SỬ DỤNG TIẾP

### 1. Tính toán sử dụng tiếp

Việc sử dụng tên trong lệnh ba địa chỉ được định nghĩa như sau:

Giả sử lệnh ba địa chỉ `i` gán một giá trị cho `x`. Nếu `x` là một toán hạng trong lệnh `j`, dòng điều khiển đi từ lệnh `i` đến `j` đồng thời dọc đường đi này ta không xen vào các lệnh gán cho `x`, thì ta nói rằng lệnh `j` dùng giá trị của `x` được tính toán tại `i`.

Trong chương này, ta chỉ đề cập đến việc tiếp tục sử dụng tên của một lệnh ba địa chỉ ngay trong khối chứa lệnh đó.

Giải thuật xác định những sử dụng tiếp theo tạo nên sự duyệt lùi đi qua các khối cơ bản. Ta có thể dễ dàng xác định lệnh ba địa chỉ cuối cùng của mỗi khối cơ bản. Vì các chương trình con có những hiệu ứng lề tùy ý nên ta giả sử rằng mỗi lời gọi chương trình con bắt đầu tại mỗi khối cơ bản mới.

Để tìm lệnh cuối cùng của một khối cơ bản, ta duyệt lùi tới phần đầu, lưu giữ các thông tin như: `x` có được tiếp tục sử dụng trong khối hay không? `x` có còn “sống” khi ra khỏi khối?

Giả sử rằng ta tới được lệnh ba địa chỉ `i:  $x := y \text{ op } z$`  trong khi duyệt lùi. Ta có thể làm theo các như sau:

1. Gán cho lệnh `i` các thông tin được tìm thấy trong bảng danh biểu. Đó là những thông tin xác định `x`, `y`, `z` có được tiếp tục sử dụng? và còn “sống”?

2. Trong bảng danh biểu, đặt `x` không “sống” và không được dùng tiếp.

3. Trong bảng danh biểu, đặt  $y, z$  “sống” và được sử dụng tiếp. Chú ý rằng các bước 2 và 3 không thể xen kẽ nhau vì  $x$  có thể là  $y$  hoặc  $z$ .

Đối với lệnh ba địa chỉ dạng  $x := y$  hoặc  $x := op\ y$ , các bước làm tương tự như trên nhưng bỏ qua  $z$ .

## 2. Bộ nhớ của các tên tạm

Nhìn chung, ta có thể gói các biến trung gian vào cùng một vị trí nhớ nếu chúng không cùng “sống”.

Hầu như tất cả các tên tạm (biến trung gian) được định nghĩa và được sử dụng trong các khối cơ bản, thông tin sử dụng tiếp có thể gói các biến trung gian. Trong chương này, ta không nói tới các biến trung gian được sử dụng trong nhiều khối.

Ta có thể cấp phát các vị trí nhớ cho các biến trung gian bằng cách kiểm tra kết quả trả về của mỗi biến và gán một biến trung gian vào vị trí đầu tiên trong trường của các biến trung gian, trường không chứa biến “sống”. Nếu một biến trung gian không được gán cho bất cứ vị trí nào được tạo ra trước đó, thêm vị trí mới vào vùng dữ liệu của chương trình con hiện hành. Trong nhiều trường hợp, các biến trung gian được gói vào trong các thanh ghi hơn là vào các vị trí nhớ.

*Chẳng hạn*, sáu biến trung gian trong khối cơ bản sau được gói vào hai vị trí nhớ là  $t_1$  và  $t_2$ :

$$t_1 := a * a$$

$$t_2 := a * b$$

$$t_2 := 2 * t_2$$

$$t_1 := t_1 + t_2$$

$$t_2 := b * b$$

$$t_1 := t_1 + t_2$$

## VI. BỘ SINH MÃ ĐƠN GIẢN

Ta giả sử rằng, bộ sinh mã này sinh mã đích từ chuỗi các lệnh ba địa chỉ. Mỗi toán tử trong lệnh ba địa chỉ tương ứng với một toán tử của máy đích. Các kết quả tính toán có thể nằm lại trong thanh ghi cho tới bao lâu có thể được và chỉ được lưu trữ khi:

(a) Thanh ghi đó được sử dụng cho sự tính toán khác

(b) Trước khi có lệnh gọi chương trình con, lệnh nhảy hoặc lệnh có nhãn.

Điều kiện (b) chỉ ra rằng bất cứ giá trị nào cũng phải được lưu vào bộ nhớ trước khi kết thúc một khối cơ bản. Vì sau khi ra khỏi khối cơ bản, ta có thể đi tới các khối khác hoặc ta có thể đi tới một khối xác định từ một khối khác. Trong trường hợp (a), ta không thể làm được điều này mà không giả sử rằng số lượng được dùng bởi khối xuất hiện trong cùng thanh ghi không có cách nào để đạt tới khối đó. Để tránh lỗi có thể xảy ra, giải thuật sinh mã đơn giản sẽ lưu giữ tất cả các giá trị khi đi qua ranh giới của khối cơ bản cũng như khi gọi chương trình con.

Ta có thể tạo ra mã phù hợp với câu lệnh ba địa chỉ  $a := b + c$  nếu ta tạo ra chỉ thị đơn ADD Rj, Ri với giá là 1. Kết quả  $a$  được đưa vào thanh ghi Ri chỉ nếu thanh ghi Ri chứa  $b$ , thanh ghi Rj chứa  $c$ , và  $b$  không được sử dụng nữa.

Nếu  $b$  ở trong  $R_i$ ,  $c$  ở trong bộ nhớ, ta có thể tạo chỉ thị:

$ADD \ c, R_i$                       giá = 2

Hoặc nếu  $b$  ở trong thanh ghi  $R_i$  và giá trị của  $c$  được đưa từ bộ nhớ vào  $R_j$  sau đó thực hiện phép cộng hai thanh ghi  $R_i, R_j$ , ta có thể tạo các chỉ thị:

$MOV \ c, R_j$

$ADD \ R_j, R_i$                       giá = 3

Qua các trường hợp trên chúng ta thấy rằng có nhiều khả năng để tạo ra mã đích cho một lệnh ba địa chỉ. Tuy nhiên, việc lựa chọn khả năng nào lại tùy thuộc vào ngữ cảnh của mỗi thời điểm cần tạo mã.

## 1. Mô tả thanh ghi và địa chỉ

Giải thuật sinh mã đích dùng bộ mô tả (descriptor) để lưu giữ nội dung thanh ghi và địa chỉ của tên.

1. Bộ mô tả thanh ghi sẽ lưu giữ những gì tồn tại trong từng thanh ghi cũng như cho ta biết khi nào cần một thanh ghi mới. Ta giả sử rằng lúc đầu, bộ mô tả sẽ khởi động sao cho tất cả các thanh ghi đều rỗng. Khi sinh mã cho các khối cơ bản, mỗi thanh ghi sẽ giữ giá trị 0 hoặc các tên tại thời điểm thực hiện.

2. Bộ mô tả địa chỉ sẽ lưu giữ các vị trí nhớ nơi giá trị của tên có thể được tìm thấy tại thời điểm thực thi. Các vị trí đó có thể là thanh ghi, vị trí trên Stack, địa chỉ bộ nhớ. Tất cả các thông tin này được lưu trong bảng danh biểu và sẽ được dùng để xác định phương pháp truy xuất tên.

## 2. Giải thuật sinh mã đích

Giải thuật sinh mã sẽ nhận vào chuỗi các lệnh ba địa chỉ của một khối cơ bản. Với mỗi lệnh ba địa chỉ dạng  $x := y \text{ op } z$  ta thực hiện các bước sau:

1. Gọi hàm `getreg` để xác định vị trí  $L$  nơi lưu giữ kết quả của phép tính  $y \text{ op } z$ .  $L$  thường là thanh ghi nhưng nó cũng có thể là một vị trí nhớ.

2. Xác định địa chỉ mô tả cho  $y$  để từ đó xác định  $y'$ , một trong những vị trí hiện hành của  $y$ . Chúng ta ưu tiên chọn thanh ghi cho  $y'$  nếu cả thanh ghi và vị trí nhớ đang giữ giá trị của  $y$ . Nếu giá trị của  $y$  chưa có trong  $L$ , ta tạo ra chỉ thị:  $MOV \ y', L$  để lưu bản sao của  $y$  vào  $L$ .

3. Tạo chỉ thị  $op \ z', L$  với  $z'$  là vị trí hiện hành của  $z$ . Ta ưu tiên chọn thanh ghi cho  $z'$  nếu giá trị của  $z$  được lưu giữ ở cả thanh ghi và bộ nhớ. Việc xác lập mô tả địa chỉ của  $x$  chỉ ra rằng  $x$  đang ở trong vị trí  $L$ . Nếu  $L$  là thanh ghi thì  $L$  là đang giữ trị của  $x$  và loại bỏ  $x$  ra khỏi tất cả các bộ mô tả thanh ghi khác.

4. Nếu giá trị hiện tại của  $y$  và/ hoặc  $z$  không còn được dùng nữa khi ra khỏi khối, và chúng đang ở trong thanh ghi thì sau khi ra khỏi khối ta phải xác lập mô tả thanh ghi để chỉ ra rằng các thanh ghi trên sẽ không giữ trị  $y$  và/ hoặc  $z$ .

Nếu mã ba địa chỉ có phép toán một ngôi thì các bước thực hiện sinh mã đích cũng tương tự như trên.

Một trường hợp cần đặc biệt lưu ý là lệnh  $x := y$ . Nếu  $y$  ở trong thanh ghi, ta phải thay đổi thanh ghi và bộ mô tả địa chỉ, là giá trị của  $x$  được tìm thấy ở thanh ghi chứa giá trị của  $y$ . Nếu  $y$  không được dùng tiếp thì thanh ghi đó sẽ không còn lưu trị của  $y$  nữa. Nếu  $y$  ở trong bộ nhớ, ta dùng hàm `getreg` để tìm một thanh ghi tải giá trị của  $y$  và xác lập rằng thanh ghi đó là

vị trí của x. Nếu ta thông báo rằng vị trí nhớ chứa giá trị của x là vị trí nhớ của y thì vấn đề trở nên phức tạp hơn vì ta không thể thay đổi giá trị của y nếu không tìm một chỗ khác để lưu giá trị của x trước đó.

### 3. Hàm getreg

Hàm getreg sẽ trả về vị trí nhớ L lưu giữ giá trị của x trong lệnh  $x := y \text{ op } z$ . Sau đây là cách đơn giản dùng để cài đặt hàm:

1. Nếu y đang ở trong thanh ghi và y sẽ không được dùng nữa sau khi thực hiện  $x := y \text{ op } z$  thì trả thanh ghi chứa y cho L và xác lập thông tin cho bộ mô tả địa chỉ của y rằng y không còn trong L.
2. Ngược lại, trả về một thanh ghi rỗng (nếu có).
3. Nếu không có thanh ghi rỗng và nếu x còn được dùng tiếp trong khối hoặc toán tử op cần thanh ghi, ta chọn một thanh ghi không rỗng R. Lưu giá trị của R vào vị trí nhớ M bằng chỉ thị MOV R,M. Nếu M chưa chứa giá trị nào, xác lập thông tin bộ mô tả địa chỉ cho M và trả về R. Nếu R giữ trị của một số biến, ta phải dùng chỉ thị MOV để lần lượt lưu giá trị cho từng biến.
4. Nếu x không được dùng nữa hoặc không có một thanh ghi phù hợp nào được tìm thấy, ta chọn vị trí nhớ của x như L.

Ví dụ 9.5: **Lệnh gán**  $d := (a - b) + (a - c) + (a - c)$

Có thể được chuyển sang chuỗi mã ba địa chỉ:

$$t := a - b$$

$$u := a - c$$

$$v := t + u$$

$$d := v + u$$

và d sẽ “sống” đến hết chương trình. Từ chuỗi lệnh ba địa chỉ này, giải thuật sinh mã vừa được trình bày sẽ tạo chuỗi mã đích với giả sử rằng: a, b, c luôn ở trong bộ nhớ và t, u, v là các biến tạm không có trong bộ nhớ.

Câu lệnh 3 địa chỉ	Mã đích	Giá	Bộ mô tả thanh ghi	Bộ mô tả địa chỉ
$t := a - b$	MOV a, R <sub>0</sub>	2	Thanh ghi rỗng, R <sub>0</sub> chứa t	t ở trong R <sub>0</sub>
	SUB b, R <sub>0</sub>	2	R <sub>0</sub> chứa t	t ở trong R <sub>0</sub>
$u := a - c$	MOV a, R <sub>1</sub>	2		u ở trong R <sub>1</sub>
	SUB c, R <sub>1</sub>	2	R <sub>1</sub> chứa u	u ở trong R <sub>1</sub>
$v := t + u$	ADD R <sub>1</sub> , R <sub>0</sub>	1	R <sub>0</sub> chứa v	v ở trong R <sub>0</sub>
$d := v + u$	ADD R <sub>1</sub> , R <sub>0</sub>	1	R <sub>1</sub> chứa u	d ở trong R <sub>0</sub>
	MOV R <sub>0</sub> , d	2	R <sub>0</sub> chứa d	d ở trong bộ nhớ

**Hình 9.9 - Chuỗi mã đích**

Lần gọi đầu tiên của hàm getreg trả về  $R_0$  như một vị trí để xác định  $t$ . Vì  $a$  không ở trong  $R_0$ , ta tạo ra chỉ thị MOV  $a, R_0$  và SUB  $b, R_0$ . Ta cập nhật lại bộ mô tả để chỉ ra rằng  $R_0$  chứa  $t$ .

Việc sinh mã đích tiếp tục tiến hành theo cách này cho đến khi lệnh ba địa chỉ cuối cùng  $d := v + u$  được xử lý. Chú ý rằng  $R_0$  là rỗng vì  $u$  không còn được dùng nữa. Sau đó ta tạo ra chỉ thị, cuối cùng của khối, MOV  $R_0, d$  để lưu biến “sống”  $d$ . Giá của chuỗi mã đích được sinh ra như ở trên là 12. Tuy nhiên, ta có thể giảm giá xuống còn 11 bằng cách thay chỉ thị MOV  $a, R_1$  bằng MOV  $R_0, R_1$  và xếp chỉ thị này sau chỉ thị thứ nhất.

#### 4. Sinh mã cho loại lệnh khác

Các phép toán xác định chỉ số và con trỏ trong câu lệnh ba địa chỉ được thực hiện giống như các phép toán hai ngôi. Hình sau minh họa việc sinh mã đích cho các câu lệnh gán:  $a := b[i]$ ,  $a[i] := b$  và giả sử  $b$  được cấp phát tĩnh.

Câu lệnh 3 địa chỉ	i trong thanh ghi $R_i$		i trong bộ nhớ $M_i$		i trên Stack	
	Mã	Giá	Mã	Giá	Mã	Giá
$a := b[i]$	MOV $b(R_i), R$	2	MOV $M_i, R$ MOV $b(R), R$	4	MOV $S_i(A), R$ MOV $b(R), R$	4
$a[i] := b$	MOV $b, a(R_i)$	3	MOV $M_i, R$ MOV $b, a(R)$	5	MOV $S_i(A), R$ MOV $b, a(R)$	5

**Hình 9.10** - Chuỗi mã đích cho phép gán chỉ mục

Với mỗi câu lệnh ba địa chỉ trên ta có thể có nhiều đoạn mã đích khác nhau tùy thuộc vào  $i$  đang ở trong thanh ghi, hoặc trong vị trí nhớ  $M_i$  hoặc trên Stack tại vị trí  $S_i$  và con trỏ trong thanh ghi  $A$  chỉ tới mẫu tin hoạt động của  $i$ . Thanh ghi  $R$  là kết quả trả về khi hàm getreg được gọi. Đối với lệnh gán đầu tiên, ta đưa  $a$  vào trong  $R$  nếu  $a$  tiếp tục được dùng trong khối và có sẵn thanh ghi  $R$ . Trong câu lệnh thứ hai ta giả sử rằng  $a$  được cấp phát tĩnh.

Sau đây là chuỗi mã đích được sinh ra cho các lệnh gán con trỏ dạng  $a := *p$  và  $*p := a$ . Vị trí nhớ  $p$  sẽ xác định chuỗi mã đích tương ứng.

Câu lệnh 3 địa chỉ	p trong thanh ghi $R_p$		p trong bộ nhớ $M_i$		p trong Stack	
	Mã	Giá	Mã	Giá	Mã	Giá
$a := *p$	MOV $*R_p, a$	2	MOV $M_p, R$ MOV $*R, R$	3	MOV $S_p(A), R$ MOV $*R, R$	3
$*p := a$	MOV $a, *R_p$	2	MOV $M_p, R$ MOV $a, *R$	4	MOV $a, R$ MOV $R, *S_p(A)$	4

**Hình 9.11** - Mã đích cho phép gán con trỏ

Ba chuỗi mã đích tùy thuộc vào  $p$  ở trong thanh ghi  $R_p$ , hoặc  $p$  trong vị trí nhớ  $M_p$ , hoặc  $p$  ở trong Stack tại offset là  $S_p$  và con trỏ, trong thanh ghi  $A$ , trỏ tới mẫu tin hoạt động của  $p$ . Thanh ghi  $R$  là kết quả trả về khi hàm getreg được gọi. Trong câu lệnh gán thứ hai ta giả sử rằng  $a$  được cấp phát tĩnh.

## 5. Sinh mã cho lệnh điều kiện

Máy tính sẽ thực thi lệnh nhảy có điều kiện theo một trong hai cách sau:

1. Rẽ nhánh khi giá trị của thanh ghi được xác định trùng với một trong sáu điều kiện sau: âm, không, dương, không âm, khác không, không dương. Chẳng hạn, câu lệnh ba địa chỉ *if x < y goto z* có thể được thực hiện bằng cách lấy x trong thanh ghi R trừ y. Sau đó sẽ nhảy về z nếu giá trị trong thanh ghi R là âm.
2. Dùng tập các mã điều kiện để xác định giá trị trong thanh ghi R là âm, bằng không hay dương. Chỉ thị so sánh CMP sẽ kiểm tra mã điều kiện mà không cần biết trị tính toán cụ thể. Chẳng hạn, CMP x, y xác lập điều kiện dương nếu  $x > y$ ,... Chỉ thị nhảy có điều kiện được thực hiện nếu điều kiện  $<, =, >, \geq, \leq, \neq$  được xác lập. Ta dùng chỉ thị nhảy có điều kiện CJ  $\leq z$  để nhảy đến z nếu mã điều kiện là âm hoặc bằng không.

Chẳng hạn, lệnh điều kiện *if x < y goto z* được dịch sang mã máy như sau.

*CMP x,y*

*CJ < z*