

<b>BỘ MÔN DUYỆT</b>	<b>ĐỀ CƯƠNG CHI TIẾT BÀI GIẢNG</b>	<b>Thay mặt nhóm</b>
<b>Chủ nhiệm Bộ môn</b>	<i>(Dùng cho tiết giảng)</i>	<b>môn học</b>
	Học phần: CHƯƠNG TRÌNH DỊCH	
	Nhóm môn học:.....	
	Bộ môn: Khoa học máy tính	
Ngô Hữu Phúc	Khoa (Viện): CNTT	Hà Chí Trung

### Thông tin về nhóm môn học

TT	Họ tên giáo viên	Học hàm	Học vị
1	Hà Chí Trung	GVC	TS
2	Võ Minh Phổ	GVC	TS
3	Nguyễn Trung Tín	TG	TS

Địa điểm làm việc: Giờ hành chính, Bộ môn Khoa học máy tính – Tầng 13 nhà S4 – Học viện Kỹ thuật Quân sự.

Địa chỉ liên hệ: Bộ môn Khoa học máy tính – Khoa Công nghệ thông tin – Học viện Kỹ thuật Quân sự. 236 Hoàng Quốc Việt.

Điện thoại, email: hct2009@yahoo.com; vominhpho@yahoo.com

### Bài giảng 01: Nhập môn Chương trình dịch

Chương I, mục:

Tiết thứ: 1-3

Tuần thứ: 1

#### - Mục đích yêu cầu

**Mục đích:** Cung cấp những thông tin về môn học, các giáo trình tài liệu liên quan, mục đích và phạm vi lý thuyết của môn học, lịch sử ra đời và các thành phần của chương trình dịch điển hình. Cơ chế và các pha làm việc của chương trình dịch.

**Yêu cầu:** sinh viên phải hệ thống lại các kiến thức cơ sở về lý thuyết ngôn ngữ lập trình, kiến thức lập trình, tự nghiên cứu và ôn tập lại những vấn đề lý thuyết khác có liên quan đến môn học như lý thuyết automata và ngôn ngữ hình thức, toán rời rạc.

- **Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

- **Thời gian:** Giáo viên giảng: 2 tiết; Thảo luận và làm bài tập trên lớp: 1 tiết; Sinh viên tự học: 6 tiết.

- **Địa điểm:** Giảng đường do P2 phân công.

- **Nội dung chính:**

- 1.1. Khái niệm về compiler
- 1.2. Vị trí của compiler trong LPS
- 1.3. Các giai đoạn làm việc của compiler
  - 1.3.1. Phân tích từ vựng (lexical analysis)
  - 1.3.2. Phân tích cú pháp (syntax analysis)
  - 1.3.3. Phân tích ngữ nghĩa (semantic analysis)
  - 1.3.4. Sinh mã trung gian (ICG)
  - 1.3.5. Tối ưu mã (code optimization)
  - 1.3.6. Sinh mã đích (code generation)
- 1.4. Vấn đề quản lý bảng ký tự
- 1.5. Xử lý lỗi biên dịch

### 1.1. Khái niệm về compiler

- **Khái niệm:** Chương trình dịch (**compiler**) là một chương trình làm nhiệm vụ đọc một chương trình được viết bằng một ngôn ngữ - ngôn ngữ nguồn (**source language - SL**) - rồi dịch nó thành một chương trình tương đương ở một ngôn ngữ khác - ngôn ngữ đích (**target language - TL**).
- Chương trình dịch là một dạng của bộ xử lý ngôn ngữ (**language processor**)

Một số khái niệm:

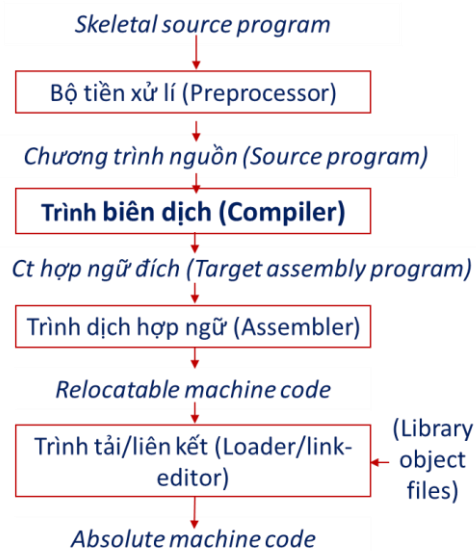
- ❖ **CTD:** chương trình dịch (**compiler**);
- ❖ **CT:** chương trình (**program**);
- ❖ **SP:** chương trình nguồn (**source program**);
- ❖ **TP:** chương trình ở ngôn ngữ đích (**target program**);
- ❖ **SL:** ngôn ngữ nguồn (source language);
- ❖ **TL:** ngôn ngữ đích (target language);
- ❖ **PL:** ngôn ngữ lập trình (programming language);
- ❖ **HLL:** ngôn ngữ bậc cao (high level language);
- ❖ **IL:** ngôn ngữ trung gian (intermediate language);
- ❖ **NL:** ngôn ngữ tự nhiên (**natural language**);
- ❖ **MC:** mã máy (machine code);
- ❖ **ML:** ngôn ngữ máy (machine language);
- Vấn đề trọng tâm:
  - ❖ Nguyên lý làm việc của các chương trình dịch;
  - ❖ Lý thuyết thiết kế ngôn ngữ lập trình (ngôn ngữ người – máy và dịch tự động);
  - ❖ Chuyển đổi từ ngôn ngữ lập trình này sang ngôn ngữ khác.

- Ứng dụng:
  - ❖ Hiểu từng ngôn ngữ, điểm mạnh điểm yếu của nó;
  - ❖ Lựa chọn ngôn ngữ và chương trình dịch thích hợp;
  - ❖ Phân biệt được công việc do **CTD** thực hiện và do **CT** ứng dụng thực hiện;
  - ❖ Thực hiện các dự án xây dựng chương trình dịch;
  - ❖ Trong giao tiếp người máy thông qua các câu lệnh
  - ❖ Áp dụng trong **NLP**, dịch tự động, tóm tắt văn bản...
- **Phân loại Compilers:** Có nhiều cách phân loại khác nhau, tùy theo tiêu chí phân loại.

Theo phương pháp dịch chạy:

- ❖ **Thông dịch:** (diễn giải - **interpreter**) đọc SP theo từng lệnh và phân tích rồi thực hiện nó (**VD:** cmd, HQTCSDL Foxpro), hoặc chuyển sang một IL + đọc CT ở IL này và thực hiện từng câu lệnh. IL được gọi là ngôn ngữ của một máy ảo (**VM**) - chương trình thông dịch thực hiện ngôn ngữ này;
- ❖ **Biên dịch (compiler):** toàn bộ chương trình nguồn được trình biên dịch chuyển sang chương trình đích ở dạng ML. Chương trình đích này có thể chạy độc lập trên máy mà không cần hệ thống biên dịch nữa.
- ❖ Theo lớp văn phạm: LL(1) (LL – Left to right, leftmost) LR(1) (LR – left to right, right most)
- **VD:** Hệ thống dịch **Java** kết hợp cả thông dịch và biên dịch. Mã nguồn **Java** được dịch ra dạng **Bytecode**. File này được một trình thông dịch gọi là **máy ảo Java** thực hiện.

## 1.2. Vị trí của compiler trong LPS



Để tạo ra một chương trình đích có khả năng thực thi (executable) thì ngoài trình biên dịch ta phải có thêm một số chương trình khác nữa.

Sơ đồ sau mô tả ngữ cảnh của một trình biên dịch trong một hệ thống xử lý ngôn ngữ (LPS: language- processing system) hay môi trường biên dịch.

### 1.3. Các giai đoạn làm việc của compiler

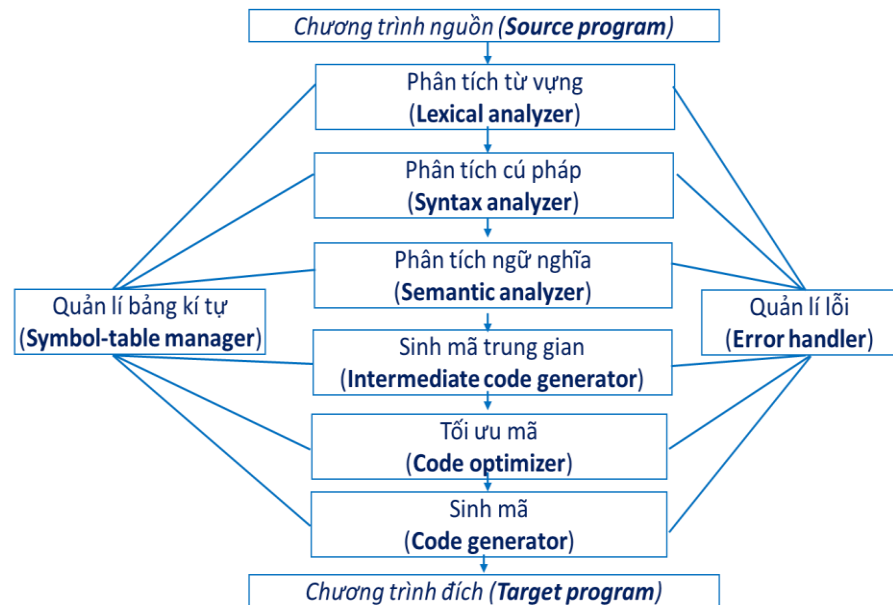
Các giai đoạn làm việc của compiler có thể phân chia theo tính logic của công việc hoặc theo thời gian làm việc.

Cấu trúc theo thời gian: lần lượt hay đồng thời.

Duyệt một lần: một số thành phần của chương trình được thực hiện đồng thời. Bộ phân tích cú pháp đóng vai trò trung tâm, điều khiển cả chương trình.

Duyệt nhiều lần: các thành phần trong chương trình được thực hiện lần lượt và độc lập với nhau. Qua mỗi một phần, kết quả sẽ được lưu vào thiết bị lưu trữ ngoài để lại được đọc vào cho bước tiếp theo.

Cấu trúc logic: 2 giai đoạn: analysis (front end) và synthesis (back end), chia làm nhiều pha làm việc.



### 1.3.1. Phân tích từ vựng (lexical analysis)

**Lexical analysis:** đọc chương trình nguồn từ trái sang phải (linear analysis/scanning) để tách ra thành các từ tổ (token);

Cũng như NL, PL được xây dựng dựa trên bộ từ vựng;

Để xây dựng một CTD, hệ thống phải tìm hiểu tập từ vựng của SL và phân tích để biết được từng loại từ vựng và các thuộc tính của nó.

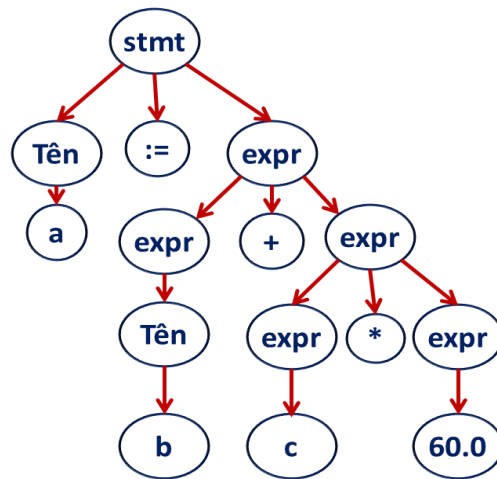
- Từ vựng trong ngôn ngữ lập trình thường được xây dựng dựa trên bộ chữ gồm có:
  - ❖ chữ cái: A .. Z, a .. z
  - ❖ chữ số: 0..9
  - ❖ các ký hiệu toán học: +, -, \*, /, (, ), =, <, >, !, %, /
  - ❖ các ký hiệu khác: [, ], ...
- Các từ vựng được ngôn ngữ hiểu bao gồm các từ khóa, các tên hàm, tên hằng, tên biến, các phép toán,...
- Các từ vựng có những qui định nhất định.
- **VD:** Câu lệnh trong chương trình nguồn viết bằng ngôn ngữ C:  $a = b + c$   
 \* 60; Chương trình phân tích từ vựng sẽ trả về:
  - ❖ a là tên (tên (định danh))
  - ❖ = là toán tử gán
  - ❖ b là tên (định danh)
  - ❖ + là toán tử cộng
  - ❖ c là định danh
  - ❖ \* là toán tử nhân
  - ❖ 60 là một số;

- Kết quả phân tích từ vựng sẽ là: (tên, a), phép gán, (tên, b) phép cộng (tên, c) phép nhân, (số, 60)

### 1.3.2. Phân tích cú pháp (syntax analysis)

- **Syntax**: Thành phần quan trọng nhất trong một ngôn ngữ. Trong **FL** thì ngôn ngữ là tập các câu thỏa mãn văn phạm (**grammar**) của ngôn ngữ đó.
- **Syntax analysis**: Phân tích cấu trúc ngữ pháp của chương trình. Các từ tố (token) của chương trình nguồn thành các cụm từ văn phạm (**grammatical phrase**) theo cấu trúc phân cấp (**syntax tree**):
  - ❖ Ngôn ngữ được định nghĩa bởi các luật sinh (**production**);
  - ❖ Phân tích cú pháp dựa vào luật sinh để xây dựng cây phân tích cú pháp (**parse tree**).
- Trong **PL**, cú pháp của nó được thể hiện bởi một bộ luật cú pháp. Bộ luật này dùng để mô tả cấu trúc của chương trình, các câu lệnh..., bao gồm:
  - ❖ các khai báo
  - ❖ biểu thức số học, biểu thức logic
  - ❖ các lệnh: lệnh gán, lệnh gọi hàm, lệnh vào ra, . . .
  - ❖ câu lệnh điều kiện if
  - ❖ câu lệnh lặp: for, while
  - ❖ chương trình con (hàm và thủ tục)
- Với một chuỗi từ tố và tập luật cú pháp của **SL**, bộ phân tích cú pháp có nhiệm vụ tự động đưa ra cây cú pháp cho chuỗi nhập; Khi cây cú pháp xây dựng xong thì quá trình phân tích cú pháp của chuỗi nhập kết thúc thành công. Ngược lại nếu áp dụng tất cả các luật nhưng không thể xây dựng được cây cú pháp của chuỗi nhập không đúng cú pháp (**Syntax error**).
- Phân tích toàn bộ **SP** thành các cấu trúc cú pháp của ngôn ngữ, từ đó để kiểm tra tính đúng đắn về mặt ngữ pháp của **SP**.
- Ví dụ: Ngôn ngữ được đặc tả bởi luật sau:
 
$$\text{Stmt} \quad \text{ten} := \text{expr}$$

$$\text{Expr} \quad \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid \text{ten} \mid \text{so}$$
- Với chuỗi nhập:  $a = b + c * 60$  ta có cây dẫn xuất như hình vẽ sau:



### 1.3.3. Phân tích ngữ nghĩa (semantic analysis)

Ngữ nghĩa: Trong phạm vi của một PL liên quan đến:

- Kiểu, phạm vi của hằng và biến (type checking)
- Phân biệt và sử dụng đúng tên hằng, tên biến, tên hàm

Phân tích ngữ nghĩa: Sử dụng cấu trúc phân cấp của giai đoạn parser để xác định các toán tử, toán hạng của các biểu thức và câu lệnh, Phân tích các đặc tính khác của chương trình mà không phải đặc tính cú pháp. Kiểm tra chương trình nguồn để tìm lỗi cú pháp và sự hợp kiểu.

CTD phải kiểm tra tính đúng đắn trong sử dụng các đại lượng. Ví dụ: không cho gán giá trị cho hằng, kiểm tra tính đúng đắn trong gán kiểu, kiểm tra phạm vi, kiểm tra sử dụng tên (tên không được khai báo trùng, dùng cho gọi hàm phải là tên có thuộc tính hàm)...

VD: Giả sử các biến rate, initial và position được khai báo là real, 60 là số integer vì vậy trình biên dịch sẽ đổi số nguyên 60 thành số thực 60.0 bằng hàm inttoREAL

### 1.3.4. Sinh mã trung gian (ICG)

- Sinh mã trung gian (**intermediate code generation**): Sinh chương trình trong **IL** nhằm mục đích:
  - ❖ dễ sinh và tối ưu mã;
  - ❖ dễ chuyển đổi về mã máy.
- Sau khi phân tích cấu trúc và ngữ nghĩa, một số trình biên dịch sẽ tạo ra một dạng biểu diễn trung gian của chương trình nguồn
- **IC** thông thường được biểu diễn dưới 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.
- **VD:** Với chuỗi nhập:  $a = b + c * 60$

sau giai đoạn phân tích thì **IC** sinh ra có dạng như sau:

```
temp1 := 60
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

### 1.3.5. Tối ưu mã (code optimitation)

**Tối ưu mã:** Sửa đổi chương trình trong **IL** nhằm cải tiến chương trình đích về hiệu năng.

**VD:** với **IC** ở trên, có thể rút gọn:

```
temp1 := id3 * 60
id1 := id2 + temp1
```

### 1.3.6. Sinh mã đích (code generation)

Sinh mã: tạo ra TP từ chương trình trong IL đã tối ưu. Thông thường là sinh ra MC hay mã hợp ngữ, do đó vấn đề quyết định là việc gán các biến cho các thanh ghi.

**VD:** Chẳng hạn CTD sử dụng các thanh ghi R1 và R2, các chỉ thị lệnh MOVF, MULF, ADDF, mã đích cho đoạn code ở trên sinh ra như sau:

```
MOVF id3, R2
MULF #60, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

## 1.4. Vấn đề quản lý bảng ký tự

Quản lý bảng ký tự: Để ghi lại các kí hiệu, tên ... đã sử dụng trong chương trình nguồn cùng các thuộc tính kèm theo như kiểu, phạm vi, giá trị... Cho việc sử dụng chúng trong suốt quá trình dịch.

Từ tổ (token) + thuộc tính (kiểu, phạm vi...) = bảng kí hiệu (symbol table)

Trong quá trình phân tích từ vựng, các tên sẽ được lưu vào bảng ký hiệu, từ giai đoạn phân tích ngữ nghĩa các thông tin khác như thuộc tính về tên (tên hằng, tên biến, tên hàm) sẽ được bổ sung trong các giai đoạn sau.

Trong các giai đoạn còn lại: lưu trữ thuộc tính của từ vựng hoặc truy xuất các thông tin thuộc tính cho từng giai đoạn.

Bảng kí hiệu được tổ chức như cấu trúc dữ liệu với mỗi phân tử là một mẫu tin dùng để lưu trữ từ vựng và các thuộc tính của nó.

Trị từ vựng: tên từ tổ.



Các thuộc tính: tầm hoạt động, số đối số, kiểu của đối số ..

### **1.5. Xử lý lỗi biên dịch**

Xử lý lỗi: Khi phát hiện ra lỗi trong quá trình dịch thì nó ghi lại vị trí gặp lỗi, loại lỗi, những lỗi khác có liên quan đến lỗi này để thông báo cho người lập trình.

Mỗi giai đoạn có thể có nhiều lỗi, tùy thuộc vào trình biên dịch mà có thể là:

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: có lỗi khi các ký tự không thể ghép thành một token (ví dụ: 15a, a@b,...)

Giai đoạn phân tích cú pháp: Có lỗi khi các token không thể kết hợp với nhau theo cấu trúc ngôn ngữ (ví dụ: if stmt then expr).

Giai đoạn phân tích ngữ nghĩa báo lỗi khi các toán hạng có kiểu không đúng yêu cầu của phép toán.

Đối với các SL, ta chỉ quan tâm đến việc sinh ra IC mà không cần biết mã máy đích của nó (không phụ thuộc vào máy đích) vì IL thường thì gần với mã máy.

#### **- Nội dung thảo luận**

1. Kinh nghiệm trong quá trình biên dịch và debug khi lập trình trong môi trường Turbo C và Visual C++.
2. Sự giống, khác nhau giữa ngôn ngữ lập trình và ngôn ngữ tự nhiên.
3. Sự giống, khác nhau giữa trình biên dịch và người biên dịch.

#### **- Yêu cầu SV chuẩn bị**

Ôn tập lại các kiến thức liên quan đến lý thuyết ngôn ngữ hình thức, automata hữu hạn và biểu thức chính quy.

#### **- Bài tập**

1. Tìm hiểu về từ tổ và cú pháp của ngôn ngữ Pascal.
2. Tìm hiểu về từ tổ và cú pháp của ngôn ngữ C.
3. Tìm hiểu về từ tổ và cú pháp của ngôn ngữ HTML, XML
4. Tìm hiểu về vấn đề biên dịch và các modul thực hiện chức năng biên dịch trong môi trường Visual Studio.net.
5. Tìm hiểu về vấn đề biên dịch và các modul thực hiện chức năng biên dịch trong môi trường Java NetBeans.

6. Tìm hiểu về cấu trúc file văn bản .RTF, .HTML, .TEX
7. Sự giống và khác nhau giữa cấu trúc văn bản với cấu trúc một chương trình trên C, Pascal?

**- Tài liệu tham khảo**

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 1.
2. **Advanced Compiler Design and Implementation.** S. Muchnick. Morgan-Kaufmann Publishers, 2007. Chương 1.
3. **Giáo trình chương trình dịch** 2<sup>nd</sup> Edition. Phạm Hồng Nguyên. NXB ĐHQG Hà Nội, 2009. Chương 1.

**- Câu hỏi ôn tập**

1. Khái niệm về ngôn ngữ, từ (chuỗi, xâu).
2. Một số phép toán cơ bản trên từ và trên ngôn ngữ.
3. Các hình thức biểu diễn ngôn ngữ. Cho ví dụ minh họa.
4. Chương trình dịch là gì? Các pha làm việc của trình biên dịch.
5. So sánh hệ thống biên dịch và thông dịch.
6. So sánh thiết kế duyệt 1 lượt và nhiều lượt.
7. Ưu điểm của kiến trúc kỳ trước và kỳ sau.
8. Tạo sao chúng ta cần sử dụng các ngôn ngữ nhân tạo?
9. So sánh chương trình dịch duyệt một lần và nhiều lần.
10. Lấy ví dụ về các chương trình biên dịch và chương trình thông dịch.

**- Ghi chú:** Các môn học tiên quyết : toán rời rạc, cấu trúc dữ liệu và giải thuật, lập trình căn bản.

## **Bài giảng 02: Lý thuyết Automata và ngôn ngữ hình thức**

Chương 2, mục:

Tiết thứ: 4-15

Tuần thứ: 2, 3, 4

**- Mục đích yêu cầu**

**Mục đích:** Sau chương này, mỗi sinh viên cần nắm vững các khái niệm sau:

- Cấu trúc ngôn ngữ tự nhiên cũng như ngôn ngữ lập trình;
- Các phép toán cơ bản trên chuỗi, ngôn ngữ;
- Cách thức biểu diễn ngôn ngữ;

- Cách phân loại văn phạm theo quy tắc của Noam Chomsky;
- Xác định các thành phần của một văn phạm;
- Mối liên quan giữa ngôn ngữ và văn phạm;
- Khái niệm ô tô mát hữu hạn, các thành phần, các dạng và sự khác biệt cơ bản giữa các dạng ô tô mát;
- Cách thức chuyển đổi tương đương giữa các dạng automata;
- Viết biểu thức chính quy ký hiệu cho tập ngôn ngữ chính quy;
- Mối liên quan giữa ô tô mát hữu hạn và biểu thức chính quy;
- Tìm các ứng dụng thực tế từ mô hình ô tô mát hữu hạn.

**Yêu cầu:** sinh viên phải hệ thống lại các kiến về chuỗi, ký hiệu, từ trong các ngôn ngữ tự nhiên như tiếng Việt, tiếng Anh; cấu trúc cú pháp của các chương trình máy tính viết bằng một số ngôn ngữ lập trình cơ bản như Pascal, C...

Để tiếp thu tốt nội dung của chương này, sinh viên cần có một số các kiến thức liên quan về lý thuyết đồ thị; hiểu các khái niệm cơ bản về kiến trúc máy tính; có sử dụng qua một số trình soạn thảo văn bản thông thường...

- **Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

- **Thời gian:** Giáo viên giảng: 6 tiết; Thảo luận và làm bài tập trên lớp: 3 tiết; Sinh viên tự học: 18 tiết.

- **Địa điểm:** Giảng đường do P2 phân công.

- **Nội dung chính:**

- 2.1. Khái niệm về ngôn ngữ, văn phạm, automata
- 2.2. Automata hữu hạn đơn định, đa định
- 2.3. Biểu thức chính quy
- 2.4. Các thuật toán biến đổi tương đương
- 2.5. Automata đẩy xuống

## 2.1. Khái niệm về ngôn ngữ, văn phạm, automata

**Khái niệm:** Văn phạm  $G$  là một bộ sắp thứ tự gồm 4 thành phần  $G = \langle \Sigma, \Delta, S, P \rangle$ , trong đó:

- $\Sigma$  - bảng chữ cái, gọi là bảng chữ cái cơ bản (bảng chữ cái kết thúc – **terminal symbol**);
- $\Delta, \Delta \cap \Sigma = \emptyset$ , gọi là bảng ký hiệu phụ (bảng chữ cái không kết thúc – **nonterminal symbol**);
- $S \in \Delta$  - ký hiệu xuất phát hay tiên đề (**start variable**);

- P - tập các luật sinh (**production rules**) dạng  $\alpha \rightarrow \beta$ ,  $\alpha, \beta \in (\Sigma \cup \Delta)^*$ , trong  $\alpha$  chứa ít nhất một ký hiệu không kết thúc (đôi khi, ta gọi chúng là các qui tắc hoặc luật viết lại).

Phân loại văn phạm theo Chomsky: **Avram Noam Chomsky** đưa ra một hệ thống phân loại các văn phạm dựa vào tính chất của các luật sinh.

**Văn phạm loại 0** – Văn phạm không hạn chế (**UG – Unrestricted Grammar**): không cần thỏa điều kiện ràng buộc nào trên tập các luật sinh;

**Văn phạm loại 1** – Văn phạm cảm ngữ cảnh (**CSG – Context Sensitive Grammar**): nếu văn phạm G có các luật sinh dạng  $\alpha \rightarrow \beta$  và  $\alpha = \alpha' A \alpha''$ ,  $A \in \Delta$ ,  $\alpha', \alpha'' \in (\Sigma \cup \Delta)^*$ ,  $|\beta| \geq |\alpha|$ ;

**Văn phạm loại 2** – Văn phạm phi ngữ cảnh (**CFG – Context-Free Grammar**): có luật sinh dạng  $A \rightarrow \alpha$  với A là một biến đơn và  $\alpha$  là chuỗi các ký hiệu thuộc  $(\Sigma \cup \Delta)^*$ ;

**Văn phạm loại 3** – Văn phạm chính quy (**RG – Regular Grammar**): có mọi luật sinh dạng tuyến tính phải hoặc tuyến tính trái.

Tuyến tính phải:  $A \rightarrow aB$  hoặc  $A \rightarrow a$ ;

Tuyến tính trái:  $A \rightarrow Ba$  hoặc  $A \rightarrow a$ ;

Với A, B là các biến đơn, a là ký hiệu kết thúc (có thể là rỗng).

Nếu ký hiệu  $L_0, L_1, L_2, L_3$  là lớp các ngôn ngữ được sinh ra bởi văn phạm loại 0, 1, 2, 3 tương ứng, ta có:  $L_3 \subset L_2 \subset L_1 \subset L_0$ .

## 2.2. Automata hữu hạn đơn định, đa định

**Automata** là một máy trừu tượng (mô hình tính toán) có cơ cấu và hoạt động đơn giản nhưng có khả năng *đoán nhận ngôn ngữ*.

**Finite automata (FA)** - mô hình tính toán hữu hạn: có khởi đầu và kết thúc, mọi thành phần đều có kích thước hữu hạn cố định và không thể mở rộng trong suốt quá trình tính toán;

Hoạt động theo từng bước rời rạc (**steps**);

Nói chung, thông tin ra sản sinh bởi một **FA** phụ thuộc vào cả thông tin vào hiện tại và trước đó. Nếu sử dụng bộ nhớ (**memory**), giả sử rằng nó có ít nhất một bộ nhớ vô hạn;

Sự phân biệt giữa các loại **automata** khác nhau chủ yếu dựa trên việc thông tin có thể được đưa vào memory hay không;

**Định nghĩa:** một DFA là một bộ năm:  $A = (Q, \Sigma, \delta, q_0, F)$ , trong đó:

1. **Q** : tập khác rỗng, tập hữu hạn các trạng thái (p, q...);
2.  **$\Sigma$**  : bộ chữ cái nhập vào (a, b, c ...);

3.  $\delta : D \rightarrow Q$ , **hàm chuyển** (hay **ánh xạ**),  $D \subseteq Q \times \Sigma$ , có nghĩa là  $\delta(p, a) = q$  hoặc  $\delta(p, a) = \emptyset$ , trong đó  $p, q \in Q$ ,  $a \in \Sigma$ ;
4.  $q_0 \in Q$  : trạng thái bắt đầu (**start state**);
5.  $F \subseteq Q$  : tập các trạng thái kết thúc (**finish states**).

Trong trường hợp  $D = Q \times \Sigma$  ta nói  $A$  là một **DFA đầy đủ**.

**Định nghĩa:** Automat hữu hạn đa định được định nghĩa bởi bộ 5:  $A = (Q, \Sigma, \delta, q_0, F)$ , trong đó:

1.  $Q$  - tập hữu hạn các trạng thái.
2.  $\Sigma$  - là tập hữu hạn các chữ cái.
3.  $\delta$  - là ánh xạ chuyển trạng thái.  $\delta: Q \times \Sigma \rightarrow 2^Q$
4.  $q_0 \in Q$  là trạng thái khởi đầu.
5.  $F \subseteq Q$  là tập trạng thái kết;

Ánh xạ  $\delta$  là một hàm đa trị (hàm không đơn định), vì vậy  $A$  được gọi là không đơn định;

**Định nghĩa:** NFA với  $\epsilon$ -dịch chuyển ( $NFA_\epsilon$ ) là bộ năm:  $A = (Q, \Sigma, \delta, q_0, F)$ , trong đó:

1.  $Q$ : tập hữu hạn các trạng thái;
2.  $\Sigma$ : tập hữu hạn các chữ cái;
3.  $\delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ ;
4.  $q_0$  là trạng thái ban đầu;
5.  $F \subseteq Q$  là tập trạng thái kết thúc.

### 2.3. Biểu thức chính quy

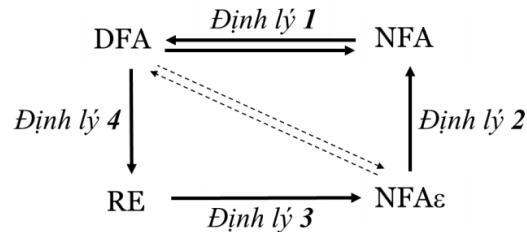
Định nghĩa: Biểu thức chính quy được định nghĩa một cách đệ quy như sau:

1.  $\epsilon$  là biểu thức chính quy.  $L(\epsilon) = \{\epsilon\}$ .  
 $\emptyset$  là biểu thức chính quy.  $L(\emptyset) = \{\emptyset\}$ .  
nếu  $a \in \Sigma$ ,  $a$  là biểu thức chính quy.  $L(a) = \{a\}$ .
2. Nếu  $r, s$  là các biểu thức chính quy thì:  
 $((r))$  là biểu thức chính quy.  $L((r)) = L(r)$ ;  
 $r+s$  là biểu thức chính quy.  $L(r+s) = L(r) \cup L(s)$ ;  
 $r.s$  là biểu thức chính quy.  $L(r.s) = L(r).L(s)$ ;  
 $r^*$  là biểu thức chính quy.  $L(r^*) = L(r)^*$ .
3. Biểu thức chính quy chỉ định nghĩa như trong 1 và 2.

**\* Tìm đọc về RE:**

1. Jeffrey E. F. Friedl. *Mastering Regular Expressions*, 2<sup>nd</sup> Edition. O'Reilly & Associates, Inc. 2002.
2. <http://www.regular-expressions.info/>

## 2.4. Các thuật toán biến đổi tương đương



**Định lý 1:** Nếu  $L$  là tập được chấp nhận bởi một NFA thì tồn tại một DFA chấp nhận  $L$ .

### Giải thuật tổng quát xây dựng DFA từ NFA:

Giả sử NFA  $A = \{Q, \Sigma, \delta, q_0, F\}$  chấp nhận  $L$ , giải thuật xây dựng DFA  $A' = \{Q', \Sigma, \delta', q_0', F'\}$  chấp nhận  $L$  như sau:

- $Q' = 2^Q$ , phần tử trong  $Q'$  được ký hiệu là  $[q_0, q_1, \dots, q_i]$  với  $q_0, q_1, \dots, q_i \in Q$ ;
- $q_0' = [q_0]$ ;
- $F'$  là tập hợp các trạng thái của  $Q'$  có chứa ít nhất một trạng thái kết thúc trong tập  $F$  của  $A$ ;
- Hàm chuyển  $\delta'([q_1, q_2, \dots, q_i], a) = [p_1, p_2, \dots, p_j]$  nếu và chỉ nếu  $\delta(\{q_1, q_2, \dots, q_i\}, a) = \{p_1, p_2, \dots, p_j\}$ .
- Đổi tên các trạng thái  $[q_0, q_1, \dots, q_i]$ .

**Định lý 2:** Nếu  $L$  được chấp nhận bởi một NFAε thì  $L$  cũng được chấp nhận bởi một NFA không có ε-dịch chuyển.

**Thuật toán:** Giả sử ta có NFAε  $A(Q, \Sigma, \delta, q_0, F)$  chấp nhận  $L$ , ta xây dựng: NFA  $A' = \{Q, \Sigma, \delta', q_0, F'\}$  như sau:

- $F' = F \cup q_0$  nếu  $\epsilon^*(q_0)$  chứa ít nhất một trạng thái thuộc  $F$ . Ngược lại,  $F' = F$ ;
- $\delta'(q, a) = \delta^*(q, a)$ .

**Hệ quả:** Nếu  $L$  là tập được chấp nhận bởi một NFAε thì tồn tại một DFA chấp nhận  $L$ .

Giải thuật xây dựng  $\delta'$  cho DFA tương đương:

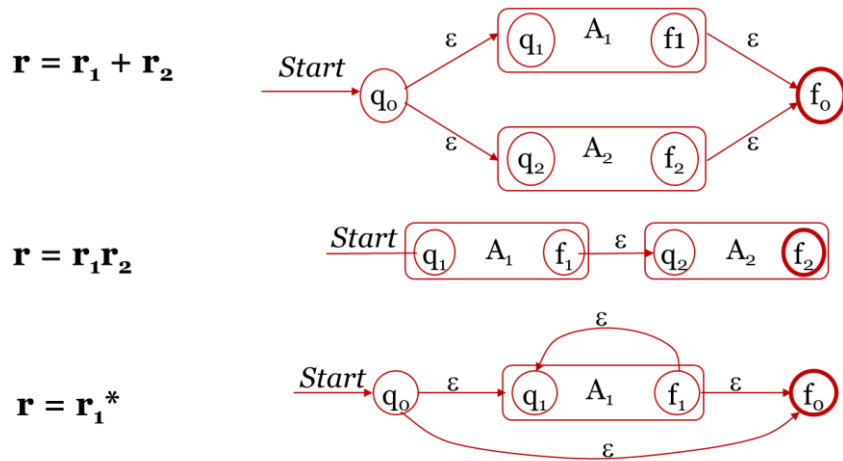
1. Tìm kiếm  $T = \epsilon^*(q_0)$ ;  $T$  chưa được đánh dấu;
2. Thêm  $T$  vào tập  $Q'$  (of DFA);
3. while (xét trạng thái  $T \in Q'$  chưa đánh dấu){
  - 3.1. Đánh dấu  $T$ ;

```

3.2.      foreach (với mỗi ký hiệu nhập a){
            U := e*(d(T,a));
            if(U không thuộc tập trạng thái Q'){
                add U to Q';
                Trạng thái U chưa được đánh dấu;
            }
            δ[T,a] = U;
        }
    }
}

```

**Định lý 3:** nếu  $r$  là RE thì tồn tại một  $NFA_\epsilon$  chấp nhận  $L(r)$ .  
(chứng minh: bài giảng, giải thuật Thompson)



**Định lý 4:** Nếu  $L$  được chấp nhận bởi một DFA, thì  $L$  được ký hiệu bởi một RE.

Chứng minh:

- ❖  $L$  được chấp nhận bởi **DFA**  $A(\{q_1, q_2, \dots, q_n\}, \Sigma, \delta, q_1, F)$
- ❖ Đặt  $R^{k}_{ij} = \{x \mid \delta(q_i, x) = q_j \text{ và nếu } \delta(q_i, y) = q_1 (y \in x) \text{ thì } 1 \leq k\}$  (có nghĩa là  $R^{k}_{ij}$  - tập hợp tất cả các chuỗi làm cho automata đi từ trạng thái  $i$  đến trạng thái  $j$  mà không đi ngang qua trạng thái nào lớn hơn  $k$ )
- ❖ Định nghĩa đệ quy của  $R^{k}_{ij}$ :

$$R^{k}_{ij} = R^{k-1}_{ik}(R^{k-1}_{kk})^*R^{k-1}_{kj} \in R^{k-1}_{ij}$$

Ta sẽ chứng minh (quy nạp theo  $k$ ) bổ đề sau: với mọi  $R^{k}_{ij}$  đều tồn tại một biểu thức chính quy ký hiệu cho  $R^{k}_{ij}$ .

- ❖  $k = 0$ :  $R^{0}_{ij}$  là tập hữu hạn các chuỗi 1 ký hiệu hoặc  $\epsilon$
- ❖ Giả sử ta có bổ đề trên đúng với  $k-1$ , tức là tồn tại RE

$$R^{k-1}_{lm} \text{ sao cho } L(R^{k-1}_{lm}) = r^{k-1}_{lm}$$

❖ Vậy đối với  $r_{ij}^k$  ta có thể chọn RE:

$$r_{ij}^k = (r_{ik}^{k-1})(r_{kk}^{k-1})^*(r_{kj}^{k-1}) + r_{ij}^{k-1}$$

→ bổ đề đã được chứng minh

nhận xét:  $L(A) = \bigcup_{q_j \in F} Rn1j$ . Vậy L có thể được ký hiệu bằng RE:

$$r = r_{1j_1}^n + r_{1j_2}^n + \dots + r_{1j_p}^n \quad \text{với } F = \{q_{j_1}, q_{j_2}, \dots, q_{j_p}\}$$

## 2.5. Automata đẩy xuống

**PDA** là một **FA** với sự bổ sung thêm một ngăn xếp (**stack**) đóng vai trò bộ nhớ, do vậy khả năng ghi nhớ của **FA** được tăng lên, dẫn đến **PDA** có khả năng đoán nhận lớp ngôn ngữ rộng hơn là **RL (RG, RE)**;

**Stack** hoạt động theo nguyên lý **FILO (LIFO)**, do đó FA sử dụng bộ nhớ có tên gọi là **Pushdown automata**;

Tại mỗi thời điểm, PDA điều khiển đồng thời cả dòng dữ liệu nhập vào (**băng nhập- tape**) và bộ nhớ - **bộ đẩy xuống (stack)**. Khi đọc một tín hiệu vào, **PDA** có thể chuyển sang một trạng thái mới, hoặc thêm, xóa đi dữ liệu từ **stack**, hoặc đồng thời cả hai;

Lớp **PDA** có khả năng đoán nhận lớp **CFL**, trong đó bao gồm các ngôn ngữ lập trình hiện đại;

**Định nghĩa:** một PDA A là một hệ thống 7 thành phần:

$$A (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

- ❖ Q : tập hữu hạn các trạng thái;
- ❖  $\Sigma$  : bộ chữ cái nhập (**input alphabet**);
- ❖  $\Gamma$  : bộ chữ cái stack (**stack alphabet**);
- ❖  $\delta$  : hàm chuyển  $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ ;
- ❖  $q_0$  : trạng thái khởi đầu;
- ❖  $Z_0$  : ký hiệu bắt đầu trên stack;
- ❖  $F \subseteq Q$  : tập các trạng thái kết thúc (nếu PDA chấp nhận chuỗi bằng Stack rỗng thì  $F = \emptyset$ ).

### - Nội dung thảo luận

Sử dụng công cụ JFLAP trong mô phỏng các thuật toán của lý thuyết ngôn ngữ hình thức và automata (<http://www.jflap.org/>)

### - Nội dung tự học

Chứng minh rằng ngôn ngữ chính quy đóng với các phép toán lấy bao đóng sao, phần bù, phép hợp, phép nối kết, phép giao, phép hiệu.



Sử dụng bổ đề bơm chứng minh tập hợp các số nguyên tố, tập hợp các số chính phương, không phải là các tập chính quy.

**- Bài tập bắt buộc**

Trong các bài tập sau, sinh viên cần:

- a) Xây dựng automata bằng đồ thị chuyển hoặc bảng chuyển để đón nhận ngôn ngữ;
- b) Đơn định hóa các automata tìm được;
- c) Chỉ ra văn phạm sinh ra các ngôn ngữ tương ứng.

1.  $L(G) = \{a^n b^m c^k \mid n, m, k > 0\}$
2.  $L(G) = \{(ab)^n (cb)^m \mid n, m \geq 0\}$
3.  $L(G) = \{(0)^n (10)^m \mid n, m \geq 0\}$
4.  $L(G) = \{\omega c \omega c \omega \mid \omega \in \{a, b\}^+\}$
5.  $L(G) = \{c^{2^n} d^n \mid n > 0\}$
6.  $L(G) = \{\omega + \omega - \omega \mid \omega \in \{a, b\}^+\}$
7.  $L(G) = \{(10)^{n-1} (01)^{n+1} \mid n > 0\}$
8.  $L(G) = \{\omega \omega^R \mid \omega \in \{0, 1\}^+\}$
9.  $L(G) = \{w \in \{a, b\}^*, n_a(w) \bmod 2 = 0, n_b(w) \bmod 2 = 1\}$   
(chẵn lần số chữ cái a và lẻ lần số chữ cái b)
10.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho không quá 2 chữ cái (a) đứng liền nhau}\}$
11.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho chỉ có duy nhất một chữ cái a}\}$
12.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho có ít nhất một chữ cái a}\}$
13.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho chỉ có không quá 3 chữ cái a}\}$
14.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho trong mỗi từ tồn tại chuỗi con dạng } a^n, n > 3\}$
15.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho trong mỗi từ tồn tại chuỗi con dạng } a^n, n < 3\}$
16.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho trong mỗi từ tồn tại không quá 2 chuỗi con dạng } a^3\}$
17.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho chữ cái đầu và cuối của mỗi từ là khác nhau}\}$
18.  $L(G) = \{w \in \{a, b\}^*, \text{sao cho trong mỗi 4 ký tự liên tiếp bất kỳ có không quá 2 chữ cái a}\}$
19.  $L(G) = \{w \in \{a, b\}^*, |w| > 3, \text{sao cho chữ cái thứ 3 khác chữ cái cuối cùng}\}$
20. Biểu diễn dạng nhị phân của các số chia hết cho 5 (ví dụ 0101, 01111)

**- Bài tập nâng cao**

1. Suy tầm các luật ngữ pháp tiếng Việt, ngôn ngữ C, Pascal. Kiểm tra xem chúng thuộc lớp văn phạm nào.
2. Trong các bài tập sau, sinh viên cần:

- a) Xây dựng automata bằng đồ thị chuyển hoặc bảng chuyển để đón nhận ngôn ngữ;
  - b) Đơn định hóa các automata tìm được;
  - c) Chỉ ra văn phạm sinh ra các ngôn ngữ tương ứng.
1. Biểu diễn các chữ số nguyên dương (1, 3, 4)
  2. Biểu diễn các chữ số nguyên (-34, +34, 34)  
Biểu diễn các số thực (dưới dạng khoa học 12.5, +12.5, -12.5, 12E3, +12E3, -12E3, 12E-3, +12E+3, -12E+3, +12E3, -12E3, 12E-312.5, +12.5, -12.5,... hoặc dạng thường 012, 0012, +012, -012, 012.5,...)
  3. Biểu diễn thời giờ gian trong ngày (21:30:58),
  4. Biểu diễn của các ngày trong năm (10/05/2010)  
Tất cả tên được đặt đúng trong ngôn ngữ C, C++ Pascal, (bao gồm các chữ cái in thường in hoa, các chữ số, chỉ bắt đầu bằng chữ cái hoặc dấu gạch dưới)
  5. Biểu diễn của các ngày trong năm (10/05/2010)  
Tất cả tên được đặt đúng trong ngôn ngữ C, C++ Pascal, (bao gồm các chữ cái in thường in hoa, các chữ số, chỉ bắt đầu bằng chữ cái hoặc dấu gạch dưới)
  6. Biểu diễn của các ngày trong năm (10/05/2010)  
Tất cả tên được đặt đúng trong ngôn ngữ C, C++ Pascal, (bao gồm các chữ cái in thường in hoa, các chữ số, chỉ bắt đầu bằng chữ cái hoặc dấu gạch dưới)

#### **- Tài liệu tham khảo**

1. **Introduction to Automata Theory, Languages, and Computation (2nd Edition).** J.E. Hopcroft, R. Motwani, J.D. Ullman. -Addison-Wesley.-2001. Chương 2, 3, 4.

#### **- Câu hỏi ôn tập**

1. Định nghĩa văn phạm, dẫn xuất và ngôn ngữ sinh bởi văn phạm. Cho ví dụ minh họa.
2. Phân loại văn phạm theo Chomsky, sự khác biệt giữa các loại văn phạm.
3. Khái niệm về văn phạm phi ngữ cảnh, khái niệm về dẫn xuất và cây dẫn xuất, sự nhập nhằng của văn phạm. Ví dụ minh họa.
4. Khái niệm về automata đẩy xuống.
5. Trình bày các phương pháp biểu diễn automata hữu hạn. Ví dụ minh họa.
6. Trình bày thuật toán đoán nhận chuỗi bởi một automata hữu hạn cho trước. Ví dụ minh họa.
7. Trình bày phương pháp biến đổi từ automata không đơn định về automata đơn định (đưa NFA về DFA).
8. Trình bày phương pháp biến đổi từ automata không đơn định có dịch chuyển-ε về automata không đơn định và không có dịch chuyển-ε (đưa  $NFA_{\epsilon}$  về NFA). Dẫn ví dụ minh họa.

9. Trình bày phương pháp biến đổi từ automata không đơn định có dịch chuyển- $\epsilon$  về automata đơn định (đưa NFA $\epsilon$  về DFA). Dẫn ví dụ minh họa.
10. Định nghĩa biểu thức chính quy. Thuật toán để xây dựng automata từ biểu thức chính quy gọi là thuật toán Thomson, trình bày thuật toán Thomson.
11. Định nghĩa biểu thức chính quy, trình bày thuật toán xây dựng biểu thức chính quy từ một automata hữu hạn cho trước.
12. Khái niệm về văn phạm chính quy. Trình bày thuật toán xây dựng một automata hữu hạn từ một văn phạm chính quy tuyến tính phải.
13. Khái niệm về văn phạm chính quy. Trình bày thuật toán xây dựng một automata hữu hạn từ một văn phạm chính quy tuyến tính trái.
14. Khái niệm về văn phạm chính quy. Trình bày thuật toán xây dựng văn phạm chính quy tuyến tính phải từ một automata hữu hạn cho trước. Ví dụ minh họa.
15. Khái niệm về văn phạm chính quy. Trình bày thuật toán xây dựng văn phạm chính quy tuyến tính trái từ một automata hữu hạn cho trước. Ví dụ minh họa.
16. Phát biểu bổ đề bơm (pumping lemma) cho tập hợp chính quy, giải thích, ý nghĩa của bổ đề bơm. Lấy ví dụ minh họa.
17. Những phép toán nào là đóng với tập hợp chính quy? Lấy ví dụ minh họa.
18. Những phép toán nào là đóng với văn phạm phi ngữ cảnh? Ví dụ minh họa.
19. Phát biểu bổ đề bơm (pumping lemma) cho tập hợp (ngôn ngữ) phi ngữ cảnh. Ví dụ minh họa.
20. Định nghĩa pushdown automata, giải thích các thành phần. Cho ví dụ minh họa.

### **Bài giảng 03: Các phương pháp phân tích từ vựng**

Chương 3, mục:

Tiết thứ: 13-18

Tuần thứ: 5, 6

#### **- Mục đích yêu cầu**

**Mục đích:** 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.

**Yêu cầu:** sinh viên phải hệ thống lại các kiến thức cơ sở về: DFA và NFA; Các automata hữu hạn đơn định và đa 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.

- **Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

- **Thời gian:** Giáo viên giảng: 4 tiết; Thảo luận và làm bài tập trên lớp: 2 tiết; Sinh viên tự học: 12 tiết.

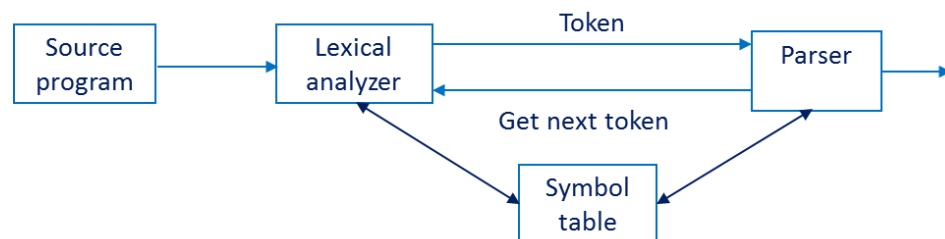
- **Địa điểm:** Giảng đường do P2 phân công.

- **Nội dung chính:**

- 3.1. Vị trí của lexical analysis
- 3.2. Các khái niệm liên quan
- 3.3. Kỹ thuật đọc chương trình nguồn
- 3.4. Nhận dạng token
  - 3.4.1. Nhận dạng các token bằng DFA
  - 3.4.2. Xây dựng DFA trực tiếp từ RE
- 3.5. Một số vấn đề trong xây dựng LA
- 3.6. Các bước và công cụ xây dựng LA
- 3.7. Bài tập thực hành

### 3.1. Vị trí của lexical analysis

LA là giai đoạn đầu tiên của quá trình dịch, giúp cho các giai đoạn biên dịch tiếp theo dễ dàng hơn (VD: giai đoạn SA không phải quan tâm đến các khoảng trắng cũng như các lời chú thích).



Nhiệm vụ chính: Đọc SP thành các *token*:

- Đọc từng ký tự một, loại bỏ các ký tự vô nghĩa (dòng trắng, space, chú thích...);
- Xác định các từ tố (token) và thông tin thuộc tính của chúng;
- Chuyển thông tin của các từ tố cho bộ parser (SA) và bảng quản lý ký hiệu (symbol-table);

- Phát hiện các lỗi cấp độ từ vựng.

### 3.2. Các khái niệm liên quan

**Lexeme:** Một nhóm các ký tự kề nhau có thể tuân theo một quy ước (mẫu hay luật) nào đó và tạo thành một từ vị.

**Pattern:** Pattern là các qui tắc kết hợp các kí tự để miêu tả một nhóm từ vị nào đó, thông thường, pattern được biểu diễn dưới dạng RE.

**String:** Là một chuỗi các kí tự từ một bảng chữ cái. Kí hiệu xâu rỗng là  $\epsilon$ , tiền tố (prefix), hậu tố (suffix), xâu con (substring), tiền tố thực sự (proper prefix)...

**Language:** Là tập hợp các xâu kí tự được xây dựng từ một bảng chữ cái cho trước.

**Token:** Một token là một tập hợp các lexemes mang một nghĩa chung xác định.

Các tokens khác nhau có các luật mô tả khác nhau. Token được mô tả bằng lời (?) bằng mẫu (pattern), hoặc các luật dưới dạng CFG (BNF) hoặc sơ đồ chuyển (FA). Ví dụ:

- Các từ khoá (keywords);
- định danh (identifiers);
- toán tử (operators);
- hằng số (consts);
- xâu kí tự (strings);
- dấu phân cách - separator(ngoặc đơn, dấu phẩy, chấm phẩy...)
- ...

**Regular definition (RD):** Một định nghĩa chính quy (**RD**) là một dãy các định nghĩa có dạng

$$\begin{aligned}d_1 &\rightarrow r_1 \\d_2 &\rightarrow r_2 \\&\dots\dots\dots \\d_n &\rightarrow r_n\end{aligned}$$

Trong đó  $d_i$  là các tên,  $r_i$  là các **RE** trên tập các kí hiệu  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$   
 Như vậy, **RD**, **RE** và **CFG (BNF)** có mối quan hệ chặt chẽ với nhau.

**VD:** **RD** của các định danh (identifiers) trong pascal là

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

**VD:** RD của các số không dấu trong pascal như 3254, 23.243E5, 16.264E-3... là

$\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$

$\text{digits} \rightarrow \text{digit} \text{ digit}^*$

$\text{optional\_fraction} \rightarrow . \text{digits} \mid \varepsilon$

$\text{optional\_exponent} \rightarrow ( E ( + \mid - \mid \varepsilon ) \text{digits} ) \mid \varepsilon$

$\text{num} \rightarrow \text{digits} \text{ optional\_fraction} \text{ optional\_exponent}$

**VD:** tìm định nghĩa (biểu thức) chính quy biểu diễn các URLs, ở các định dạng khác nhau:

*http://www.yahoo.com/*

*http://yahoo.com, yahoo.com*

*E:\tkprasad\cs680\asg-R00\asg2.html*

*file:///C:/JavaSources/jdk1\_3-src/README.html*

*telnet://gamma.cs.wright.edu*

*ftp://www.cs.wright.edu*

*ftp://tkprasad@www.cs.wright.edu/*

- **Tìm đọc về RE:** Jeffrey E. F. Friedl. *Mastering Regular Expressions*, 2<sup>nd</sup> Edition. O'Reilly & Associates, Inc. 2002.

### 3.3. Kỹ thuật đọc chương trình nguồn

**Mục đích:** để làm tăng tốc độ xử lý SP bộ LA không đọc từng ký tự hay từng dòng một mà đọc cả block vào buffer. Để xác định các từ vị thông thường dùng 2 kỹ thuật:

- ❖ Kỹ thuật cặp bộ đệm (buffer pairs);
- ❖ Kỹ thuật cầm canh.

**Kỹ thuật cặp bộ đệm:** Chia buffer thành 2 nửa, mỗi nửa chứa n kí tự ( n = 1024, 4096, ...). Sử dụng 2 con trỏ dò tìm trong buffer:

- ❖ p1: (lexeme beginning) Đặt tại vị trí đầu của một từ vị.
- ❖ p2: (forward): di chuyển trên từng kí tự trong buffer để xác định từ tố.

**Kỹ thuật cặp bộ đệm:**

```
if (p2 ở ranh giới một nửa bộ đệm) {  
    đọc n ký hiệu từ SP vào nửa bên phải; p2 := p2 + 1;  
}  
else if (p2 ở tận cùng bên phải bộ đệm) {  
    đọc tiếp N ký hiệu từ SP vào nửa bên trái bộ đệm.  
    chuyển p2 về ký tự tận cùng bên trái của bộ đệm.
```

```

}
else p2 = p2 + 1;

```

#### Phương pháp cầm canh:

```

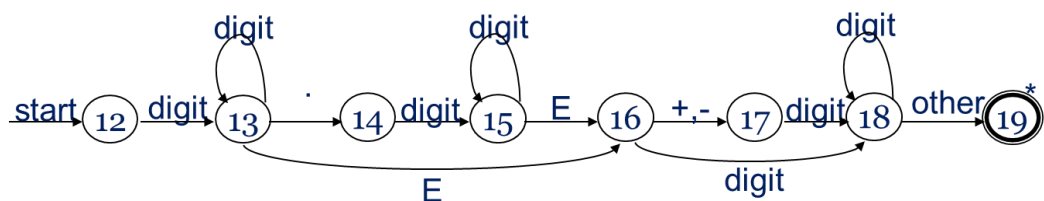
p2 := p2 + 1;
if (p2 ≠ eof) {
    if (p2 ở ranh giới một nửa buffer) {
        lấp đầy n ký hiệu nhập vào nửa bên phải buffer; p2 = p2 +
1;
    }
    else if (p2 ở tận cùng bên phải buffer){
        lấp đầy N ký hiệu vào nửa bên trái buffer; chuyển p2 về đầu buffer;
    }
    else /* dừng sự phân tích từ vựng*/

```

### 3.4. Nhận dạng token

#### 3.4.1. Nhận dạng các token bằng DFA

- Các bước xây dựng LA nhận dạng tokens bằng DFA:
  - a) Tập hợp tất cả các mẫu của từ tố;
  - b) Lập bộ phân tích từ vựng bằng phương pháp diễn giải đồ thị chuyển, sử dụng các lệnh lựa chọn if hoặc switch
  - c) Kết hợp các đồ thị chuyển thành một đồ thị chuyển duy nhất. Lập bộ phân tích từ vựng điều khiển bằng bảng (mô phỏng ô tô máy hữu hạn đơn định).
- **Ưu điểm:** dễ hiểu, dễ viết.
- **Nhược điểm:** gắn kết cấu đồ thị chuyển vào trong chương trình. Khi thay đổi đồ thị thì phải viết lại chương trình nên khó bảo trì.
- **VD:** FA đoán nhận các unsigned numbers trong pascal:
  - sốthựcmũ  $\rightarrow$  (chữsố)<sup>+</sup> [(chữsố)<sup>+</sup>] ? [ E [ +|- ] ? (chữsố)<sup>+</sup> ] ?
  - sốthực  $\rightarrow$  chữsố<sup>+</sup> . chữsố<sup>+</sup>
  - sốnguyên  $\rightarrow$  chữsố<sup>+</sup>



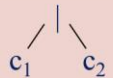
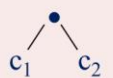

#### 3.4.2. Xây dựng DFA trực tiếp từ RE

- Biến đổi trực tiếp RE về DFA không thông qua NFA:

- Trước hết, ta đánh dấu biểu thức cần biến đổi bằng ký hiệu đặc biệt nào đó, chẳng hạn #.

$$r \rightarrow (r)\# \quad \text{augmented RE}$$

- Sau đó chúng ta tạo ra cây cú pháp (**syntax tree**) cho biểu thức gia tố:
  - ❖ tất cả các ký hiệu kết thúc (gồm cả # và  $\epsilon$ ) trong RE đã cho sẽ nằm ở các lá;
  - ❖ các nodes bên trong sẽ chứa các toán tử trong biểu thức;
  - ❖ Mỗi một ký hiệu kết thúc (gồm cả #) sẽ được đánh thứ tự;
  - ❖ Xác định hàm **followpos** cho từng node lá.
- **followpos(i)** -- tập hợp các vị trí mà có thể đứng ở sau vị trí i trong biểu thức gia tố. **followpos** chỉ định nghĩa cho node lá, không cho các node trong.
- Để tính được hàm **followpos**, chúng ta cần 2 hàm hỗ trợ cho các node, bao gồm cả các nút trong của cây cú pháp.
  - ❖ **firstpos(n)** -- tập hợp các vị trí của những ký tự đầu tiên trong các xâu được tạo bởi cây con với đỉnh là n.
  - ❖ **lastpos(n)** -- tập hợp các vị trí của những ký hiệu cuối cùng trong các xâu được sinh bởi biểu thức con với đỉnh là n.
  - ❖ **nullable(n)** -- true nếu xâu rỗng nằm trong số những xâu sinh bởi biểu thức con với đỉnh là n, ngược lại thì là false.
- Bảng tóm tắt cách tính các hàm firstpos, lastpos, nullable:

$n$	<b>nullable(n)</b>	<b>firstpos(n)</b>	<b>lastpos(n)</b>
leaf labeled $\epsilon$	true	$\emptyset$	$\emptyset$
leaf labeled with position i	false	{i}	{i}
	nullable( $c_1$ ) or nullable( $c_2$ )	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
	nullable( $c_1$ ) and nullable( $c_2$ )	if (nullable( $c_1$ )) $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ else firstpos( $c_1$ )	if (nullable( $c_2$ )) $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else lastpos( $c_2$ )
	true	firstpos( $c_1$ )	lastpos( $c_1$ )

- **followpos(n)** được tính theo các quy tắc sau:
  1. Nếu n là node kết nối với con bên trái là  $c_1$  và con phải  $c_2$ , và i là một vị trí trong **lastpos**( $c_1$ ), khi đó tất cả các vị trí trong **firstpos**( $c_2$ ) cũng thuộc về **followpos**(i).



2. Nếu **n** là node bao đóng sao (\*), và **i** là một vị trí trong **lastpos(n)**, khi đó tất cả các vị trí trong **firstpos(n)** cũng thuộc về **followpos(i)**.
- Nếu **firstpos** và **lastpos** được tính cho từng node, **followpos** cho mỗi vị trí có thể tính bởi một lần duyệt theo chiều sâu của syntax tree.
- Giải thuật xây dựng DFA từ RE:
  1. *Create the syntax tree of (r) #*
  2. *Calculate the functions: followpos, firstpos, lastpos, nullable*
  3. *Put **firstpos(root)** into the states of DFA as an unmarked state.*
  4. *while (there is an unmarked state S in the states of DFA) do*
    - ❖ *mark S*
    - ❖ *for each input symbol a do*
      - *let  $s_1, \dots, s_n$  are positions in S & symbols in those positions are a*
      - $S' \leftarrow \text{followpos}(s_1) \cup \dots \cup \text{followpos}(s_n)$
      - $\text{move}(S, a) \rightarrow S'$
      - *if ( $S'$  is not empty and not in the states of DFA)*
  - put  $S'$  into the states of DFA as an unmarked state.*
  5. *Trạng thái bắt đầu DFA là firstpos(root), trạng thái kết thúc là các nhãn có chứa #*

### 3.5. Một số vấn đề trong xây dựng LA

Bỏ qua chú thích (comment):

- ❖ Thông thường, ta không phân tích các comment thành token, bởi vậy LA trả về token tiếp theo mà không phải là chú thích cho SA;
- ❖ Như vậy comments chỉ được xử lý trong bộ LA, và chúng không làm phức tạp thêm cú pháp của PL.

Symbol table:

- ❖ symbol table lưu giữ thông tin về các token;
- ❖ Làm thế nào để quản lý và sử dụng bảng ký tự? Những thuật toán nào được sử dụng? (Bảng băm (hash table), thêm token vào bảng băm, tìm vị trí của token theo lexeme).
- ❖ Vị trí của token trong SP? (cho vấn đề error handling).

### 3.6. Các bước và công cụ xây dựng LA

Sơ đồ chung để xây dựng một bộ phân tích từ vựng:

1. *Sưu tầm tất cả các luật từ vựng, các luật này thường được mô tả bằng lời.*

2. Vẽ đồ thị chuyển cho từng luật một. Trước đó có thể mô tả chúng bằng các biểu thức chính quy để tiện theo dõi và chỉnh sửa, và làm dễ cho việc dựng đồ thị.
3. Kết hợp các luật này thành một đồ thị chuyển duy nhất.
4. Chuyển đồ thị này thành bảng chuyển của automata.
5. Bổ sung các thành phần chương trình để thành bộ phân tích hoạt động được.
6. Thêm phần báo lỗi để thành bộ phân tích từ vựng hoàn chỉnh.

Một số công cụ có sẵn cho phép xây dựng một bộ phân tích từ vựng dựa trên các biểu thức chính quy như Lex, Flex, Tplex, Jlex...

### **- Nội dung thảo luận**

Cú pháp của ngôn ngữ lập trình C, Pascal, Java, XML...

- C: [http://www.cs.man.ac.uk/~pjj/bnf/c\\_syntax.bnf](http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf)
- C++: <http://www.nongnu.org/hcb/>
- JAVA: <http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html>
- XML: <http://cm.bell-labs.com/cm/cs/who/wadler/xml/>

Cài đặt các công cụ LEX, FLEX, Gold Parser. Sử dụng các công cụ này cho việc thiết kế bộ phân tích từ vựng của các ngôn ngữ trên.

- **LEX/YACC/FLEX/Bison:** <http://dinosaur.compilertools.net/>
- **Gold Parser:** <http://www.devincook.com/GOLDParser/index.htm>

### **- Yêu cầu SV chuẩn bị**

Hệ thống lại các kiến thức cơ sở về: DFA và NFA; Các automata hữu hạn đơn định và đa 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.

### **- Bài tập bắt buộc**

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

a) Pascal;      b) C;

2. Xác định các từ vựng và từ tố trong các đoạn chương trình sau:

a) PASCAL

```
function max(i, j: integer): integer;
```

```
begin
```

```
  if i>j then max := i;
```

```
else max := j;
```

```
end;
```

b) C:

```
int max( int i, j){
```

```
    return i>j?i:j;
```

```
}
```

3. Mô tả các ngôn ngữ được biểu diễn bằng các biểu thức chính quy sau:

a)  $0(0+1)^*0$

b)  $((\varepsilon + 0)^*1^*)^*$

c)  $(0+1)^*0(0+1)(0+1)$

d)  $0^*10^*10^*10^*$

4) Viết các biểu thức chính quy hoặc định nghĩa chính quy cho các ngôn ngữ sau:

a) các chuỗi chữ cái tiếng Anh có chứa 5 nguyên âm theo thứ tự

b) Các xâu chữ cái đúng thứ tự từ điển

c) Các chú thích trong C;

d) Các chuỗi ký tự không có ký tự nào lặp lại

e) Các chuỗi ký tự có ít nhất một ký tự lặp lại;

f) Các chuỗi nhị phân có chứa chẵn lần số 0 và lẻ lần số 1;

g) Các chuỗi số nhị phân không chứa chuỗi con 001;

h) Các chuỗi số nhị phân không chứa chuỗi hậu tố 001;

5. Thiết kế văn phạm cho các ngôn ngữ sau, xác định ngôn ngữ nào là ngôn ngữ chính quy:

a) Tập các chuỗi nhị phân mà sau mỗi số 0 có ít nhất một số 1;

b) Tập các chuỗi nhị phân có số chữ số 0 đúng bằng số chữ số 1;

c) Tập các chuỗi nhị phân có số chữ số 0 khác số chữ số 1;

d) Các chuỗi số nhị phân không chứa chuỗi con 001;

e) Các chuỗi số nhị phân không chứa chuỗi hậu tố 001;

6. Xây dựng các DFA bằng đồ thị chuyển và bảng chuyển để đoán nhận các ngôn ngữ sau. Trình bày các bước chuyển thực hiện trong quá trình xử lý chuỗi “ababbab”

a)  $(a+b)^*$

b)  $(a^*+b^*)^*$

c)  $((\varepsilon + a)b^*)^*$

d)  $(a+b)^*abb(a+b)^*$

e)  $(a+b)^*a(a+b)$

f)  $(a+b)^*a(a+b)(a+b)$

g)  $(a+b)^*a(a+b)(a+b)(a+b)$

### - Bài tập nâng cao

1. Áp dụng bổ đề bơm, bạn hãy chứng minh ngôn ngữ sau đây không là ngôn ngữ chính quy:

$$a) \{a^m b^n \mid 0 \leq m < n\}.$$

$$e) \{a^m b^n \mid 0 \leq n < m\}.$$

$$b) \{a^m b^n c^n d^m \mid m \geq 0, n \geq 0\}.$$

$$f) \{a^m b^m c^n d^n \mid m \geq 0, n \geq 0\}.$$

$$c) \{a^m b^n c^p d^q \mid m + n = p + q\}.$$

$$g) \{a^m b^n c^k \mid m = n \text{ or } m = k\}.$$

$$d) \{a^n b^m c^{2n} \mid m \geq 0, n \geq 0\}.$$

2. Cho L là ngôn ngữ chính quy:

a)  $L^R$  có phải ngôn ngữ chính quy không?

b)  $\text{Pref}(L)$  có phải ngôn ngữ chính quy không?

c)  $\text{Suf}(L)$  có phải ngôn ngữ chính quy không?

d)  $\text{Suf}(\text{pref}(L))$  có phải ngôn ngữ chính quy không?

3. Viết chương trình mô phỏng hoạt động của FA.

4. Viết chương trình sinh tự động bộ phân tích từ vựng dựa trên luật sinh của văn phạm.

#### **- Tài liệu tham khảo**

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 5.
2. **Introduction to Automata Theory, Languages, and Computation (2nd Edition).** J.E. Hopcroft, R. Motwani, J.D. Ullman. -Addison-Wesley.- 2001. Chương 4.
3. **Mastering Regular Expressions,** 2<sup>nd</sup> Edition. Jeffrey E. F. Friedl. O'Reilly & Associates, Inc. 2002. Chương 2.

#### **- Câu hỏi ôn tập**

1. Biến đổi NFAs, NFA sang DFA (xem bài giảng 03 – Ngôn ngữ hình thức);
2. Biến đổi DFA sang RG và ngược lại (xem bài giảng 04 – Ngôn ngữ hình thức);
3. Xây dựng FA từ biểu thức chính quy bằng giải thuật Thomson (xem bài giảng 03 –NNHT);
4. Biến đổi trực tiếp từ biểu thức chính quy sang DFA (bài giảng số 02- Chương trình dịch);
5. Biến đổi từ DFA sang biểu thức chính quy (xem bài giảng 03 – Ngôn ngữ hình thức);

## **Bài kiểm tra: Lý thuyết ngôn ngữ hình thức và automata hữu hạn**

Tiết thứ: 19-21

Tuần thứ: 7

### **- Mục đích yêu cầu**

**Mục đích:** Đánh giá kiến thức sinh viên về NNHT và automata hữu hạn.

**Yêu cầu:** Nghiêm túc, trung thực.

**- Hình thức tổ chức dạy học:** Kiểm tra

**- Thời gian:** 3 tiết.

**- Địa điểm:** Giảng đường do P2 phân công.

**- Nội dung chính:** Kiểm tra, đánh giá

**- Yêu cầu SV chuẩn bị:** sinh viên phải hệ thống lại các kiến thức cơ sở về: Automata hữu hạn (FA), Văn phạm phi ngữ cảnh (Context Free Grammar – CFG), Automat đẩy xuống (Pushdown Automata – PDA); Cách biến đổi từ một RG về một DFA; Cách biến đổi từ một CFG về một PDA;

**- Tài liệu tham khảo:**

## **Bài giảng 04: Các phương pháp phân tích cú pháp**

Chương 4, mục:

Tiết thứ: 22-27

Tuần thứ: 8, 9

### **- Mục đích yêu cầu**

**Mục đích:** Trang bị cho học viên những kiến thức về: 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.

**Yêu cầu:** sinh viên phải hệ thống lại các kiến thức cơ sở về: Automata hữu hạn (FA), Văn phạm phi ngữ cảnh (Context Free Grammar – CFG), Automat đẩy xuống (Pushdown Automata – PDA); Cách biến đổi từ một RG về một DFA; Cách biến đổi từ một CFG về một PDA;

**- Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

**- Thời gian:** Giáo viên giảng: 4 tiết; Thảo luận và làm bài tập trên lớp: 2 tiết; Sinh viên tự học: 12 tiết.

**- Địa điểm:** Giảng đường do P2 phân công.

**- Nội dung chính:**

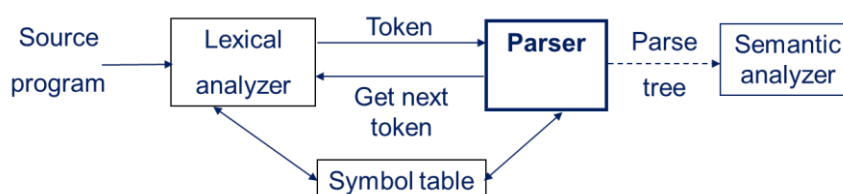
4.1. Vị trí của phân tích cú pháp

4.2. Một số vấn đề trong phân tích cú pháp

- 4.2.1. Văn phạm phi ngữ cảnh
- 4.2.2. Phân tích cú pháp từ trên xuống
- 4.2.3. Một số chiến lược phục hồi lỗi
- 4.2.4. Phân tích cú pháp từ dưới lên

#### 4.3. Công cụ xây dựng SA

### 4.1. Vị trí của phân tích cú pháp



- Cú pháp của một **PL** có thể được miêu tả bởi một **CFG**. Thông thường người ta sử dụng dạng **BNF (Backus-Naur Form)** để diễn đạt chúng.
- Nhiệm vụ chính của SA (Syntax Analyzer, parser):
  - ❖ kiểm tra xem **SP** của chương trình có thỏa mãn cú pháp của ngôn ngữ hay không, bằng cách nhận chuỗi các **token** từ bộ phân tích từ vựng và xác định chuỗi đó có được sinh ra bởi văn phạm của **SL** không.
  - ❖ Thông thường, parser sẽ chỉ ra cấu trúc cú pháp của mã nguồn, cấu trúc này trong hầu hết trường hợp có thể biểu diễn như là một parse tree.
  - ❖ Ngược lại, trả về các thông báo lỗi.
- Các chiến lược phân tích cú pháp:
  - ❖ Phân tích từ trên xuống (**top-down parsing**): Cây cú pháp được tạo từ trên xuống, bắt đầu từ gốc;
  - ❖ Phân tích từ dưới lên (**bottom-up parsing**): Cây cú pháp được tạo từ dưới lên, bắt đầu từ các lá.
- Cả hai dạng parsers đều quét luồng dữ liệu vào từ trái qua phải (tại mỗi thời điểm thường là 1 token).
- **Vấn đề:** Trong quá trình biên dịch xuất hiện nhiều lỗi do đó **SA** phải phát hiện và thông báo lỗi chính xác cho người lập trình đồng thời không làm chậm những chương trình được viết đúng.
  - Trên thực tế, những parsers hiệu quả chỉ có thể ứng dụng cho một lớp con **CFGs**:
  - Phân tích đệ quy ( $O(c^n)$ );
  - thuật toán phân tích **CYK (Coke-Younger-Kasami)** ( $O(n^3)$ );
  - (thuật toán phân tích **Earley**) ( $O(n^3)$  hoặc  $O(n^2)$  hoặc  $O(n)$ );

- **LL(k)** đối với top-down parsing ( $O(n)$ );
- **LR(k)** đối với bottom-up parsing ( $O(n)$ ).

## 4.2. Một số vấn đề trong phân tích cú pháp

### 4.2.1. Văn phạm phi ngữ cảnh

- Để định nghĩa cấu trúc của **PL** ta dùng **CFG**:
  - ❖ Chỉ rõ các đặc điểm cú pháp của một **PL**;
  - ❖ Thiết kế văn phạm với ngôn ngữ đã cho;
  - ❖ Có thể chuyển đổi từ một CFG sang một bộ parser nhờ một số công cụ có sẵn (..!).
- **VD**: CFG sau định nghĩa các biểu thức số học đơn giản:
 
$$E \rightarrow E A E \mid (E) \mid -E \mid id$$

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

Trong đó E, A là các kí tự chưa kết thúc (E còn là kí tự bắt đầu), các kí tự còn lại là các kí tự kết thúc

- **Một số khái niệm liên quan**: dẫn xuất trực tiếp, gián tiếp trái nhất, phải nhất.

Văn phạm phi ngữ cảnh (**CFG – Context-Free Grammar**): luật sinh có dạng  $A \rightarrow \alpha$  với A là một biến đơn và  $\alpha$  là chuỗi các ký hiệu thuộc  $(\Sigma \cup \Delta)^*$ ;

**Tính nhập nhằng của văn phạm (ambiguity)**: Một văn phạm sinh ra nhiều hơn một **parse tree** cho một câu được gọi là văn phạm nhập nhằng (mơ hồ, đa nghĩa). Nói cách khác một văn phạm nhập nhằng sẽ sinh ra nhiều hơn một dẫn xuất trái nhất hoặc dẫn xuất phải nhất cho cùng một câu.

- ❖  $E \rightarrow E+E \rightarrow id+E \rightarrow id+E*E \rightarrow id+id*E \rightarrow id+id*id$
- ❖  $E \rightarrow E*E \rightarrow E+E*E \rightarrow id+E*E \rightarrow id+id*E \rightarrow id+id*id$
- **Văn phạm đệ quy trái**: Một văn phạm được gọi là đệ quy trái (left recursion) nếu tồn tại một dẫn xuất có dạng  $A \xrightarrow{+} A\alpha$  (trong đó A là 1 kí hiệu chưa kết thúc,  $\alpha$  là một xâu).
- Các phương pháp phân tích từ trên xuống không thể xử lí văn phạm đệ quy trái, do đó cần phải biến đổi văn phạm để loại bỏ các đệ quy trái.
- Đệ quy trái có hai loại:
  - ❖ Loại trực tiếp: Có dạng  $A \rightarrow A\alpha$
  - ❖ Loại gián tiếp: Gây ra do dẫn xuất của hai hoặc nhiều bước
- VD**:  $S \rightarrow Aa \mid b$ ;  $A \rightarrow Sc \mid d$
- Loại bỏ đệ quy trái trực tiếp:
  - ❖ Ta nhóm các luật sinh thành

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

trong đó  $\beta_1 \dots \beta_n$  không bắt đầu với A

❖ Thay luật sinh trên bởi các luật sinh sau:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

- Loại bỏ đệ qui trái gián tiếp:

1. Sắp xếp các ký hiệu không kết thúc theo thứ tự  $A_1, A_2, \dots, A_n$

2. Áp dụng thuật toán sau:

for i:=1 to n do {

    for j:=1 to i -1 do {

        Thay luật sinh dạng  $A_i \rightarrow A_j \gamma$  bởi

$$A_i \rightarrow \beta_1 \gamma \mid \beta_2 \gamma \mid \dots \mid \beta_k \gamma, \text{ trong đó}$$

$$A_j \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_k$$

    }

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

};

#### 4.2.2. Phân tích cú pháp từ trên xuống

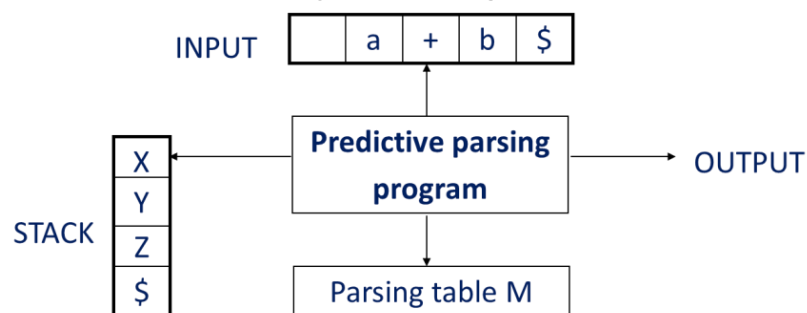
Top-down parsing cố gắng tìm ra dẫn xuất trái nhất của chương trình nguồn.

Bottom-up parsing cố tìm ra cây dẫn xuất phải nhất của chương trình nguồn.

- **Top-down parser:** Cây cú pháp được tạo từ gốc tới lá. Có một số kỹ thuật SA từ trên xuống như:
  - ❖ SA đệ quy lùi (Recursive-Descent parsing);
  - ❖ SA đoán trước (Predictive parsing);
  - ❖ SA đoán trước đệ quy (recursive predictive parsing);
  - ❖ SA đoán trước không đệ quy (non-recursive predictive parsing);
  - ❖ ...
- Recursive-Descent Parsing:
  - ❖ Backtracking (nếu lựa chọn luật sinh này không thỏa mãn thì quay lui thử áp dụng luật sinh khác).
  - ❖ Kỹ thuật tổng quát, nhưng ko được sử dụng rộng rãi;
  - ❖ Không hiệu quả...
- Predictive Parsing:  $O(n)$ 
  - ❖ Không quay lui, nhưng đòi hỏi văn phạm phải được **thừa số hóa trái** (*left-factored*);



- ❖ Chỉ áp dụng cho một lớp con của CFG là văn phạm LL(k) (Left-to-right parse, Leftmost-derivation, k-symbol lookahead);
- **Phép thừa số hóa trái** (*left-factoring*) là phép biến đổi CFG để có được một văn phạm thuận tiện cho việc phân tích dự đoán.
- **Ý tưởng:** khi không rõ luật sinh nào trong hai luật sinh có thể dùng để khai triển một ký hiệu chưa kết thúc A, ta có thể viết lại các A-luật sinh nhằm "hoãn" lại việc quyết định cho đến khi thấy đủ yếu tố cho một lựa chọn đúng.
- **SA** đoán trước không đệ quy còn gọi là LL(1) parser hoặc **table-driven parser**(phân tích dựa trên bảng) hoạt động theo mô hình sau:



- Bộ phân tích cú pháp được điều khiển bởi **Predictive parsing program**
- **Input buffer:** Dòng dữ liệu cần phân tích. Ở cuối ta thêm vào một ký hiệu đặc biệt \$.
- **Output:** Luật sinh được áp dụng trong từng bước dẫn xuất (left-most derivation).
- **Stack:**
  - ❖ Các ký hiệu của văn phạm;
  - ❖ Stack lúc khởi tạo chỉ chứa \$ và ký hiệu bắt đầu S.
  - ❖ Khi stack rỗng (i.e. chỉ còn \$), quá trình phân tích kết thúc.
- **Parsing table**
  - ❖ Mảng hai chiều  $M[A, a]$ , mỗi hàng là 1 ký hiệu chưa kết thúc, mỗi cột là 1 ký hiệu kết thúc hoặc là ký hiệu đặc biệt \$
  - ❖ Mỗi ô chỉ chứa không quá 1 luật sinh.
- **Thuật toán phân tích:** Stack (S\$) và bộ đệm chứa chuỗi nhập dạng w\$, con trỏ ip trở tới ký hiệu đầu tiên của w\$;
 

```

while (X ≠ $)
  X = top (Stack) và ip trở đến a;
  if (X là ký hiệu kết thúc hoặc $)
    if (X = a) pop(X) và dịch chuyển ip;
    else error ( ) ;      //gọi chương trình phục hồi lỗi;
      
```

```

else if (  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  ) { //  $X$  là non-terminal ;
    pop( $X$ );
    push  $Y_k, Y_{k-1}, \dots, Y_1$  vào Stack;
    Xuất ra luật sinh  $X \rightarrow Y_1 Y_2 \dots Y_k$ ;
}
else error ( ) /* Stack rỗng */
}

```

- Với chuỗi nhập có dạng id+id:

Stack	input	output
\$E	id+id\$	$E \rightarrow TE'$
\$E'T	id+id\$	$T \rightarrow FT'$
\$E'T'F	id+id\$	$F \rightarrow id$
\$E'T'id	id+id\$	
\$E'T'	+id\$	$T' \rightarrow \epsilon$
\$E'	+id\$	$E' \rightarrow +TE'$
\$E'T+	+id\$	
\$E'T	id\$	$T \rightarrow FT'$
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	
\$E'T'	\$	$T' \rightarrow \epsilon$
\$E'	\$	$E' \rightarrow \epsilon$
\$	\$	accept

### Xây dựng bảng phân tích cho văn phạm LL(1):

- Hàm **FIRST** và **FOLLOW**: Là các hàm xác định các tập hợp cho phép xây dựng bảng phân tích M và phục hồi lỗi theo chiến lược panic-mode.
  - ❖ Nếu  $\alpha$  là một xâu thì  $FIRST(\alpha)$  là tập hợp các ký hiệu kết thúc mà nó có thể bắt đầu ở một chuỗi được dẫn xuất từ  $\alpha$ . Nếu  $\alpha \Rightarrow^* \epsilon$  thì  $\epsilon$  thuộc  $FIRST(\alpha)$
  - ❖ Nếu A là một kí hiệu chưa kết thúc thì  $FOLLOW(A)$  là tập các kí hiệu kết thúc mà nó có thể xuất hiện ngay bên phải A trong một dẫn xuất từ S. Nếu  $S \Rightarrow^* \alpha A$  thì  $\$$  thuộc  $FOLLOW(A)$
- Quy tắc xác định hàm FIRST:
  - ❖ Nếu X là kí hiệu kết thúc thì  $FIRST(X)$  là  $\{X\}$
  - ❖ Nếu  $X \rightarrow \epsilon$  là một luật sinh thì thêm  $\epsilon$  vào  $FIRST(X)$ .

- ❖ 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  $\epsilon$  của  $\text{FIRST}(Y_1)$  vào  $\text{FIRST}(X)$ .
- ❖ Nếu  $\epsilon \in \text{FIRST}(Y_1)$  thì tiếp tục thêm vào  $\text{FIRST}(X)$  tất cả các ký hiệu kết thúc khác  $\epsilon$  của  $\text{FIRST}(Y_2)$ .
- ❖ Nếu  $\epsilon \in \text{FIRST}(Y_1) \cap \text{FIRST}(Y_2)$  thì thêm tất cả các ký hiệu kết thúc khác  $\epsilon \in \text{FIRST}(Y_3) \dots$

Cuối cùng thêm  $\epsilon$  vào  $\text{FIRST}(X)$  nếu  $\epsilon \in \bigcap_{i=1}^k \text{FIRST}(Y_i)$

- **Qui tắc tính các tập hợp FOLLOW** (chỉ áp dụng cho ký hiệu chưa kết thúc)

1. Đặt  $\$$  vào  $\text{FOLLOW}(S)$  ( $S$  là kí hiệu bắt đầu)
2. Nếu  $A \rightarrow \alpha B \beta$  thì mọi phần tử thuộc  $\text{FIRST}(\beta)$  ngoại trừ  $\epsilon$  đều thuộc  $\text{FOLLOW}(B)$
3. Nếu  $A \rightarrow \alpha B$  hoặc  $A \rightarrow \alpha B \beta$  và  $\beta \Rightarrow^* \epsilon$  thì mọi phần tử thuộc  $\text{FOLLOW}(A)$  đều thuộc  $\text{FOLLOW}(B)$
4. áp dụng các quy tắc trên đến khi không thể thêm gì vào các tập FOLLOW

- Văn phạm không phải là LL(1):

- ❖ Văn phạm đệ quy trái không thể là LL(1) grammar.

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

➔ mọi ký hiệu kết thúc trong  $\text{FIRST}(\beta)$  cũng có mặt trong  $\text{FIRST}(A\alpha)$  vì rằng  $A\alpha \Rightarrow \beta\alpha$ .

➔ Nếu  $\beta$  là  $\epsilon$ , mọi ký hiệu kết thúc trong  $\text{FIRST}(\alpha)$  cũng có trong  $\text{FIRST}(A\alpha)$  và  $\text{FOLLOW}(A)$ .

- ❖ Văn phạm chưa thừa số hóa trái, không thể là LL(1) grammar

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

➔ mọi ký hiệu kết thúc trong  $\text{FIRST}(\alpha\beta_1)$  cũng có trong  $\text{FIRST}(\alpha\beta_2)$ .

- ❖ Văn phạm nhập nhằng không thể là LL(1) grammar.

➔ **Tính chất của văn phạm LL(1):** Văn phạm  $G$  là LL(1) khi và chỉ khi những điều kiện sau thỏa mãn đối với 2 luật phân biệt bất kỳ  $A \rightarrow \alpha$  và  $A \rightarrow \beta$

- ❖ Cả  $\alpha$  và  $\beta$  không dẫn ra từ các xâu có ký hiệu đầu giống nhau.
- ❖ Tối đa chỉ có 1:  $\alpha$  hay  $\beta$  có thể dẫn tới  $\epsilon$ .

- ❖ Nếu  $\beta$  dẫn tới  $\varepsilon$ , thì  $\alpha$  không thể dẫn ra xâu bắt đầu bằng ký hiệu trong FOLLOW(A).

#### 4.2.3. Một số chiến lược phục hồi lỗi

- Trong SA, sẽ rất bất tiện nếu chương trình dừng và thông báo lỗi ngay 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**.
- Phần lớn việc phát hiện và phục hồi lỗi trong một trình biên dịch tập trung vào giai đoạn SA.
- Bộ xử lý lỗi (**error handler**) trong compiler:
  - ❖ 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.
- Lỗi có thể xảy ra trong phân tích dự đoán (LL(1) parsing)
  - ❖ Nếu ký hiệu kết thúc ở đỉnh stack không khớp với ký hiệu đang xét.
  - ❖ Nếu đỉnh của stack là một ký hiệu chưa kết thúc A, còn ký hiệu đang xét là a, trong khi đó trên bảng phân tích ô M[A,a] là rỗng.
- Khi đó, chương trình dịch phải xử lý thế nào?
  - ❖ Đưa ra một thông báo lỗi (càng chính xác và rõ ràng càng tốt về lỗi xảy ra).
  - ❖ Đánh dấu lỗi này và tiếp tục phân tích phần còn lại.
- Phương thức “hoảng sợ” (Panic-Mode Error Recovery):
  - ❖ Khi một lỗi được phát hiện thì bộ phân tích cú pháp bỏ qua từng ký hiệu một cho đến khi tìm thấy một token đồng bộ (synchronizing token);
  - ❖ các token đồng bộ của ký hiệu phụ A là tất cả các ký hiệu kết thúc trong tập **follow** của A;
- **VD:** Một cách phục hồi lỗi đơn giản theo phương thức hoảng sợ trong LL(1) parsing:
  - ❖ Tất cả các ô trống được đánh dấu là **synch** để chỉ ra rằng parser sẽ bỏ qua mọi ký hiệu cho đến khi gặp ký hiệu trong tập **follow(A)** trên đỉnh stack.
  - ❖ Để xử lý ký hiệu kết thúc không khớp, chương trình dịch loại nó ra khỏi stack và đưa ra thông báo lỗi (rằng ký hiệu đó bị thay thế).
- Chiến lược mức ngữ đoạn (Phrase-Level Error Recovery):

- ❖ Khi phát hiện một lỗi, SA 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 để có thể tiếp tục. Chẳng hạn, dấu phẩy (,) bởi dấu chấm phẩy (;), xóa một dấu phẩy lạ hoặc thêm vào một dấu chấm phẩy.
- ❖ Mỗi ô trống trong bảng ký hiệu được lấp đầy bởi con trỏ tới một thủ tục xử lý lỗi.
- Các thủ tục xử lý lỗi theo chiến lược mức ngữ đoạn có thể là:
  - ❖ Thay đổi, chèn, hoặc xóa bỏ các ký tự nhập;
  - ❖ Đưa ra thông báo lỗi (gần đúng);
  - ❖ Loại bỏ một số phần tử (pop) ra khỏi stack.
- **Lưu ý:** Việc lập các thủ tục không đúng có thể dẫn đến việc lặp vô hạn quá trình xử lý.
- Dùng các luật sinh sửa lỗi (Error-Productions):
  - ❖ 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;
  - ❖ Tuy nhiên phương pháp này không thực tế.
- Chiến lược hiệu chỉnh toàn cục (Global-Correction):
  - ❖ Trong trường hợp lý tưởng, chương trình dịch thực hiện ít thay đổi, chèn, xóa nhất có thể trong việc hiệu chỉnh lỗi.
  - ❖ Phân tích một cách toàn cục chuỗi nhập: 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..
  - ❖ Đây là phương pháp đang nghiên cứu và chưa thực tế.

#### 4.2.4. Phân tích cú pháp từ dưới lên

- **Bottom-up parser** là phương pháp phân tích tạo cây cú pháp của mã nguồn từ lá lên gốc (tìm dẫn xuất phải nhất theo thứ tự ngược lại dẫn xuất, bắt đầu từ chuỗi nhập).
- Phân tích bottom-up (phân tích phải) là dừng khi và chỉ khi  $G$  không chứa suy dẫn  $A \Rightarrow^+ A$  và không có sản xuất  $B \rightarrow \epsilon$  (sản xuất rỗng).
- Phân tích từ dưới lên tổng quát còn được biết đến với tên gọi **phân tích đẩy thu (Shift-Reduce Parser)**, vì hai hành động cơ bản của nó là **shift** (dịch chuyển, đẩy) và **reduce** (rút gọn).

$\omega \Leftarrow \dots \Leftarrow S$

- **VD:** Cho văn phạm:

$$S \rightarrow a A B e$$

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

$$B \rightarrow d$$

- Câu **abcde** có thể thu gọn về S theo các bước sau:

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

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

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

$$a A B e$$

$$S$$

- Đảo ngược lại quá trình trên ta thu được dẫn xuất phải nhất:

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

**Handles:** Handle của một right-sentential form  $\gamma$  là một luật sinh  $A \rightarrow \beta$ , một xâu  $\alpha$  sao cho  $\gamma = \alpha\beta\omega$  và  $S \Rightarrow_{rm}^* \alpha A \omega \Rightarrow_{rm} \alpha\beta\omega$ . Đôi khi ta còn gọi  $\beta$  là một handle, xâu  $\omega$  bên phải  $\beta$  chỉ chứa các kí hiệu kết thúc

Nếu một văn phạm là không mơ hồ thì với mỗi right-sentential form có duy nhất một handle của nó. Ví dụ, trong dẫn xuất:

$$S \Rightarrow_{rm} \underline{aABe} \Rightarrow_{rm} aA\underline{de} \Rightarrow_{rm} a\underline{Abcde} \Rightarrow_{rm} a\underline{b}bcde$$

các handle được gạch chân

Dẫn xuất phải nhất của một xâu có thể nhận được theo thứ tự ngược bằng thuật toán **handle-pruning** (cắt tia handle) được miêu tả như sau:

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

1. Bắt đầu từ  $\gamma_n$ , tìm handle  $A_n \rightarrow \beta_n$  trong  $\gamma_n$ , và thay thế  $\beta_n$  trong  $\gamma_n$  bởi  $A_n$ , thu được  $\gamma_{n-1}$ .

2. Tìm  $A_{n-1} \rightarrow \beta_{n-1}$  trong  $\gamma_{n-1}$ , và thay thế  $\beta_{n-1}$  bởi  $A_{n-1}$ , nhận được  $\gamma_{n-2}$ .

....

3. Lặp lại các bước trên đến khi thu được S.

Bảng sau cho thấy quá trình phân tích từ dưới lên với xâu vào  $id_1 + id_2 * id_3$

Stack	Input	Action
\$	$id_1 + id_2 * id_3$ \$	shift
\$ $id_1$	$+ id_2 * id_3$ \$	reduce by $E \rightarrow id$
\$ E	$+ id_2 * id_3$ \$	shift
\$ E +	$id_2 * id_3$ \$	shift
\$ E + $id_2$	$* id_3$ \$	reduce by $E \rightarrow id$
\$ E + E	$* id_3$ \$	shift
\$ E + E *	$id_3$ \$	

\$ E + E * id <sub>3</sub>	\$	shift
\$ E + E * E	\$	reduce by $E \rightarrow id$
\$ E + E	\$	reduce by $E \rightarrow E * E$
\$ E	\$	reduce by $E \rightarrow E + E$
		accept

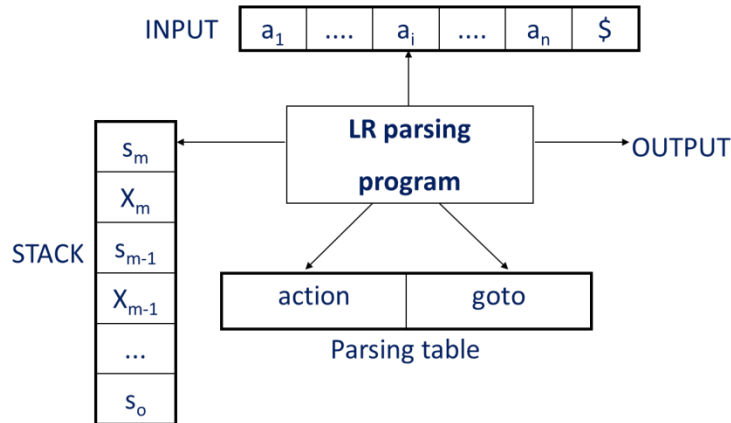
Có 4 hành động có thể xảy ra trong shift-parser:

1. **Shift**: Ký tự kế tiếp được đẩy lên đỉnh stack;
2. **Reduce**: Thu gọn handle ở đỉnh stack bằng ký hiệu chưa kết thúc;
3. **Accept**: phân tích thành công.
4. **Error**: Phát hiện lỗi cú pháp, gọi tiến trình phục hồi lỗi.

Lúc khởi tạo stack chỉ chứa ký hiệu đánh dấu \$. Chuỗi vào cũng được đánh dấu ở cuối bởi ký hiệu \$.

- Một phương pháp tổng quát hơn của kỹ thuật Shift - Reduce là phân tích cú pháp LR (LR parsing: left-to-right, rightmost derivation).
  - ❖ A shift-reduce parser cố gắng thu gọn chuỗi nhập về ký hiệu S.
  - ❖ Tại mỗi bước rút gọn, chuỗi con của dòng ký hiệu nhập so khớp với vế phải của luật sinh nào đó sẽ thay thế bởi ký hiệu chưa kết thúc bên vế trái của luật sinh đó.
  - ❖ Nếu xâu con được chọn đúng, thì dẫn xuất phải nhất của chuỗi nhập sẽ được tạo ra theo thứ tự ngược lại.
- LR(k) là một kỹ thuật phân tích cú pháp từ dưới lên hiệu quả, có thể sử dụng để phân tích một lớp rộng các văn phạm phi ngữ cảnh.
  - ❖ L(Left-to-right): Duyệt chuỗi nhập từ trái sang phải
  - ❖ R(Rightmost derivation): Xây dựng chuỗi dẫn xuất phải nhất đảo ngược
  - ❖ k: Số lượng ký hiệu lookahead tại mỗi thời điểm, dùng để đưa ra quyết định phân tích. Khi không đề cập đến k, chúng ta hiểu ngầm là  $k = 1$
- Ưu điểm của LR:
  - ❖ Có thể nhận biết hầu như tất cả các ngôn ngữ lập trình được tạo ra bởi văn phạm phi ngữ cảnh
  - ❖ Phương pháp phân tích cú pháp LR là phương pháp tổng quát của phương pháp shift-reduce không quay lui
  - ❖ Lớp văn phạm có thể dùng phương pháp LR là một lớp rộng lớn hơn lớp văn phạm có thể sử dụng phương pháp dự đoán

- ❖ Có thể xác định lỗi cú pháp nhanh ngay trong khi duyệt dòng nhập từ trái sang phải
- Nhược điểm của LR:
  - ❖ Xây dựng LR parser khá phức tạp
- Mô hình của LR parser:



- STACK lưu chuỗi  $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$  trong đó  $s_m$  nằm trên đỉnh STACK một ký hiệu văn phạm,  $s_i$  là một trạng thái tóm tắt thông tin chứa trong STACK bên dưới nó
- Parsing table bao gồm 2 phần : Hàm **action** và hàm **goto**
  - ❖  $\text{action}[s_m, a_i]$  có thể có một trong 4 giá trị :
    1. **shift s**: Đẩy  $s$ , trong đó  $s$  là một trạng thái
    2. **reduce**: Thu gọn bằng luật sinh  $A \rightarrow \beta$
    3. **accept**: Chấp nhận
    4. **error**: Báo lỗi
  - ❖  $\text{goto}$  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  $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$ . Cấu hình biểu diễn right-sentential form  $X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$
- Sự thay đổi cấu hình theo hàm action như sau:
  - ❖ Nếu  $\text{action}[s_m, a_i] = \text{shift } s$ , cấu hình chuyển thành  $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$ , trong đó  $s = \text{action}[s_m, a_i]$
  - ❖ Nếu  $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$ , cấu hình chuyển thành  $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$ , trong đó  $s = \text{goto}[s_{m-r}, A]$ ,  $r = |\beta| = |X_{m-r+1} \dots X_m|$
  - ❖ Nếu  $\text{action}[s_m, a_i] = \text{accept}$ , quá trình phân tích thành công
  - ❖ Nếu  $\text{action}[s_m, a_i] = \text{error}$ , gọi thủ tục phục hồi lỗi



- **VD:** Xét văn phạm cho các phép toán số học + và \* sau:

$$(1) E \rightarrow E + T \quad (2) E \rightarrow T \quad (3) T \rightarrow T * F$$

$$(4) T \rightarrow F \quad (5) F \rightarrow (E) \quad (6) F \rightarrow id$$

- **Ý nghĩa :**  $s_i$ : đẩy  $s_i$ ;  $r_j$ : thu gọn bởi luật  $j$ ; **acc**: chấp nhận; **blank**: error

State	Action						Goto		
	Id	+	*	(	)	\$	E	T	F
<b>0</b>	$s_5$			$s_4$			1	2	3
<b>1</b>		$s_6$				<b>acc</b>			
<b>2</b>		$r_2$	$s_7$		$r_2$	$r_2$			
<b>3</b>		$r_4$	$r_4$		$r_4$	$r_4$			
<b>4</b>	$s_5$			$s_4$			8	2	3
<b>5</b>		$r_6$	$r_6$		$r_6$	$r_6$			
<b>6</b>	$s_5$			$s_4$				9	3
<b>7</b>	$s_5$			$s_4$					10
<b>8</b>		$s_6$			$s_{11}$				
<b>9</b>		$r_1$	$s_7$		$r_1$	$r_1$			
<b>10</b>		$r_3$	$r_3$		$r_3$	$r_3$			
<b>11</b>		$r_5$	$r_5$		$r_5$	$r_5$			

- Xây dựng bảng phân tích SLR
  - ❖ Có 3 phương pháp xây dựng một bảng phân tích cú pháp LR từ văn phạm là Simple LR (SLR), Canonical LR và Lookahead-LR (LALR), các phương pháp khác nhau về tính hiệu quả cũng như tính dễ cài đặt
  - ❖ Phương pháp SLR, là phương pháp yếu nhất nếu tính theo số lượng văn phạm có thể xây dựng thành công, nhưng đây lại là phương pháp dễ cài đặt nhất
  - ❖ Một văn phạm có thể xây dựng được SLR parser được gọi là một văn phạm SLR
  - ❖ Giải thuật xây dựng họ tập hợp các mục LR(0) (kí hiệu là C) của văn phạm  $G'$

*procedure* Item ( $G'$ )

*begin*

$C := \{closure(\{ S' \rightarrow .S \})\};$

*repeat*

*For  $I \in C$ ,  $X$  sao cho  $goto(I, X) \neq \Phi$  và  $goto(I, X) \notin C$  thì add  $goto(I, X)$  to  $C$ ;*

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

*end;*

### **Xây dựng SLR parsing table**

1. Xây dựng họ tập hợp các mục của  $G'$ :  $C = \{ I_0, I_1, \dots, I_n \}$
  2. Trạng thái  $i$  được xây dựng từ  $I_i$ . Các action tương ứng trạng thái  $i$  xác định như sau:
    - a) Nếu  $A \rightarrow \alpha.a\beta \in I_i$  và  $goto(I_i, a) = I_j$  thì  $action[i, a] = \text{"shift } j\text{"}$ ,  $a$  là ký hiệu kết thúc
    - b) Nếu  $A \rightarrow \alpha. \in I_i$  thì  $action[i, a] = \text{"reduce } (A \rightarrow \alpha)\text{"}$ , với mọi  $a \in FOLLOW(A)$ ,  $A \neq S'$
    - c) Nếu  $S' \rightarrow S \cdot \in I_i$  thì  $action[i, \$] = \text{"accept"}$ .
- Nếu một action đưng độ được sinh ra bởi các luật trên, ta nói văn phạm không phải là SLR(1). Giải thuật thất bại
3. Nếu  $goto(I_i, A) = I_j$  thì  $goto[i, A] = j$ ,  $A$  là kí hiệu chưa kết thúc
  4. 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  $S' \rightarrow \cdot S$

### **4.3. Công cụ xây dựng SA**

Công cụ phân tích cú pháp YACC, BISON (Yet Another Compiler - Compiler): công cụ (trong UNIX) cho phép xây dựng bộ phân tích cú pháp một cách tự động;

Yacc được tạo bởi S. C. Johnson vào những năm đầu của thập kỉ 70, sử dụng phương pháp LALR.

#### **- Nội dung thảo luận**

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.

#### **- Yêu cầu SV chuẩn bị**

Cài đặt các một số công cụ phân tích cú pháp và sinh bộ phân tích cú pháp tự động cho một số văn phạm đơn giản (một phần của C, hoặc Pascal).

**- Bài tập bắt buộc**

1. Cho văn phạm  $S \rightarrow SbS \mid bSaS \mid \varepsilon$

- Chứng minh rằng văn phạm này là nhập nhằng;
- Xây dựng các dẫn xuất phải nhất và trái nhất cho xâu vào “abab”;
- Phân tích Top-down cho xâu trên

2. Cho văn phạm:

$$E \rightarrow TX; X \rightarrow +E \mid \varepsilon; T \rightarrow (E) \mid aY; Y \rightarrow *T \mid \varepsilon$$

Phân tích xâu vào «  $a^*a+a$  » bằng phương pháp LL(1).

3. Cho văn phạm  $S \rightarrow AS \mid BA; A \rightarrow aB \mid \varepsilon; B \rightarrow bA \mid \varepsilon$ .

- Xây dựng bảng phân tích LL(1).
- Hỏi văn phạm trên có phải văn phạm LL(1) không?

4. Cho văn phạm  $S \rightarrow (L) \mid a; L \rightarrow L,S \mid S$ .

- Xác định ngôn ngữ sinh bởi văn phạm trên;
- Khử đệ quy trái;
- Xây dựng bộ phân tích cú pháp LR cho văn phạm;
- Dùng bộ phân tích cú pháp đã được xây dựng để vẽ cây phân tích cú pháp cho các chuỗi nhập sau:

i) (a,a)                      ii) (a,(a,a))                      (a,(a,a),(a,a)))

- Xây dựng dẫn xuất trái nhất, phải nhất cho từng chuỗi ở phần d.

5. Cho văn phạm chứa các luật sản xuất sau:

$$\text{bexpr} := \text{bexpr} \textbf{ or } \text{bterm} \mid \text{bterm}$$

$$\text{bterm} := \text{bterm} \textbf{ and } \text{bfactor} \mid \text{bfactor}$$

$$\text{bfactor} := \textbf{not} \text{bfactor} \mid (\text{bexpr}) \mid \text{true} \mid \text{false}$$

- Hãy xây dựng bảng phân tích LR(1) cho văn phạm G.
- Xây dựng cây phân tích cú pháp cho chuỗi: **not (true or false)**
- Chứng minh rằng văn phạm này sinh ra toàn bộ các biểu thức

Boole;

- Văn phạm trên có phải văn phạm nhập nhằng không? Tại sao?
- Xây dựng bộ phân tích cú pháp SLR cho văn phạm.

**- Bài tập nâng cao**

- Viết chương trình tính FIRST và FOLLOW.
- Ứng dụng chương trình trên để phát hiện văn phạm có phải LL(1) không?
- Cài đặt bộ phân tích LL(1).
- Cài đặt bộ phân tích LR(1).
- Cài đặt chương trình lập bảng SLR.

6. Cài đặt chương trình lập bảng LR chuẩn.
7. Cài đặt chương trình lập bảng LALR.
8. So sánh hiệu quả phân tích của các bộ phân tích LL(1) và LR(1).

**- Tài liệu tham khảo**

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 4
2. **Modern Compiler Implementation in C.** A.W. Appel - Cambridge University Press, 1997. Chương 3.
3. **Engineering a Compiler.** K. Cooper, L. Torczon. - Morgan-Kaufman Publishers, 2005. Chương 3.

**- Câu hỏi ôn tập**

1. Input và output của phân tích cú pháp là gì?
2. Tại sao cây phân tích nhận được từ thuật toán phân tích topdown được gọi là cây phân tích trái?
3. Chứng minh rằng văn phạm đệ qui trái không thể là LL(1).
4. Chứng minh rằng văn phạm LL(1) không thể nhập nhằng.
5. Khử đệ quy trái trong văn phạm CFG (xem bài giảng 03 – chương trình dịch);
6. Lập bảng phân tích cho văn phạm LL(1) (xem bài giảng 03 – chương trình dịch);
7. Nêu những điểm giống nhau và khác nhau của thuật toán phân tích bottom-up và thuật toán phân tích LR? (Nêu ngắn gọn, không trình bày lại thuật toán)
8. Những văn phạm như thế nào thì không phân tích được bằng thuật toán bottom-up? Tại sao?
9. Tại sao lớp văn phạm LR(k) lại rộng hơn lớp văn phạm LL(k)?

**Bài giảng 05: Phân tích ngữ nghĩa – biên dịch dựa cú pháp**

Chương I, mục:

Tiết thứ: 28-33

Tuần thứ: 10, 11

**- Mục đích yêu cầu**

**Mục đích:** 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. Bài 5 trình bày các cách biểu diễn ngữ nghĩa của một chương trình. Khi học xong bài này sinh viên cần 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.

**Yêu cầu:** Sinh viên cần ôn tập lại và nắm chắc các kiến thức bài 3, 4.

**- Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

**- Thời gian:** Giáo viên giảng: 4 tiết; Thảo luận và làm bài tập trên lớp: 2 tiết; Sinh viên tự học: 12 tiết.

**- Địa điểm:** Giảng đường do P2 phân công.

**- Nội dung chính:**

5.1. Mục tiêu, đầu vào, đầu ra của semantic analysis.

5.2. Biên dịch dựa cú pháp

5.2.1. Các khái niệm cơ bản

5.2.2. Xây dựng cây cú pháp

5.2.3. Lược đồ dịch.

5.2.4. Thiết kế bộ dịch dự đoán

5.3. Các vấn đề về kiểm tra kiểu

5.4. Vấn đề xử lý tối nghĩa

**5.1. Mục tiêu, đầu vào, đầu ra của semantic analysis.**

**5.2. Dịch dựa cú pháp**

- Quá trình dịch được điều khiển theo cấu trúc cú pháp của chương trình nguồn.
- Cú pháp-điều khiển (syntax-directed) là cơ chế điều khiển bám theo cấu trúc cú pháp
- Được dùng cho mọi khối sau khối phân tích cú pháp
- Cách đơn giản nhất: gắn thêm luật xử lý (luật ngữ nghĩa) vào luật cú pháp
- Cần cơ chế duyệt các luật ngữ nghĩa dựa theo cây phân tích.

**5.2.1. Các khái niệm cơ bản**

Định nghĩa trực tiếp cú pháp (syntax- directed definition) 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 (attribute)

Các thuộc tính có thể là một xâu, một số, một kiểu dữ liệu, một địa chỉ trong bộ nhớ...

Giá trị các thuộc tính được tính bởi các luật ngữ nghĩa (semantic rule) đi kèm. Mỗi luật ngữ nghĩa được viết như lời gọi các thủ tục hoặc một đoạn chương trình

Cây phân tích cú pháp có trình bày giá trị các thuộc tính tại mỗi nút gọi là cây chú thích

Trong một định nghĩa trực tiếp cú pháp, mỗi luật sinh  $A \rightarrow \alpha$  kết hợp một tập luật ngữ nghĩa có dạng  $b := f(c_1, c_2, \dots, c_k)$  trong đó  $f$  là một hàm và:

1)  $b$  là một thuộc tính tổng hợp (synthesized attribute) 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 (inherited attribute) 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

**Ví dụ 1:** Định nghĩa trực tiếp cú pháp (ĐNTTCP) cho một máy tính đơn giản

PRODUCTION	SYMANATIC RULES
$L \rightarrow E n$	<code>print(E.val)</code>
$E \rightarrow E_1 + T$	<code>E.val := E<sub>1</sub>.val + T.val</code>
$E \rightarrow T$	<code>E.val := T.val</code>
$T \rightarrow T_1 * F$	<code>T.val := T<sub>1</sub>.val * F.val</code>
$T \rightarrow F$	<code>T.val := F.val</code>
$F \rightarrow (E)$	<code>F.val := E.val</code>
$F \rightarrow \text{digit}$	<code>F.val := digit.lexval</code>

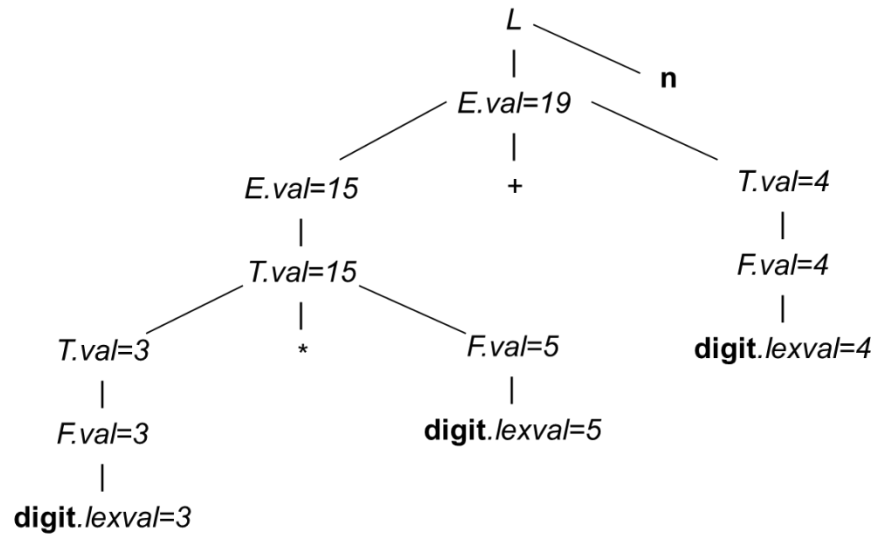
Token digit có thuộc tính tổng hợp *lexval* mà giá trị được cung cấp bởi bộ phân tích từ vựng

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 (S- attributed definition)

Trong cây phân tích cú pháp của định nghĩa S- thuộc tính, các luật ngữ nghĩa tính giá trị các thuộc tính cho các nút từ dưới lên, từ lá đến gốc

**Ví dụ 2:** ĐNTTCP trong ví dụ 1 là định nghĩa S- thuộc tính. Cây chú thích cho biểu thức  $3*5+4n$  (n kí hiệu cho newline) như sau:



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 nút anh em của nó.

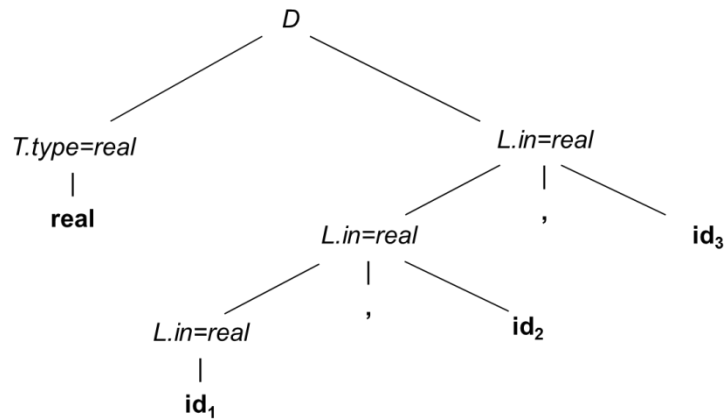
Nói chung ta có thể viết một định nghĩa trực tiếp cú pháp thành một định nghĩa S- thuộc tính. Nhưng trong một số trường hợp, việc sử dụng thuộc tính kế thừa lại thuận tiện vì tính tự nhiên của nó.

**Ví dụ 3:** Xét định nghĩa trực tiếp cú pháp sau cho sự khai báo kiểu cho biến

PRODUCTION	SYMANTIC RULES
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in;$
$L \rightarrow \text{id}$	$\text{addtype} \quad (\text{id.entry}, L.in)$
	$\text{addtype} \quad (\text{id.entry}, L.in)$

Các luật kết hợp với luật sinh của L gọi thủ tục *addtype* dùng để nhập kiểu cho mục vào của định danh trong symbol table

**Ví dụ 4:** Cây chú thích cho dòng lệnh `real id1, id2, id3;` như sau

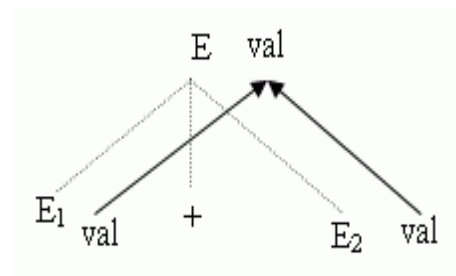


Đồ thị phụ thuộc (dependency graph): Trong 1 cây cú pháp có thể chứa cả thuộc tính tổng hợp và thuộc tính kế thừa, ta dùng đồ thị phụ thuộc để biểu diễn các loại thuộc tính đó

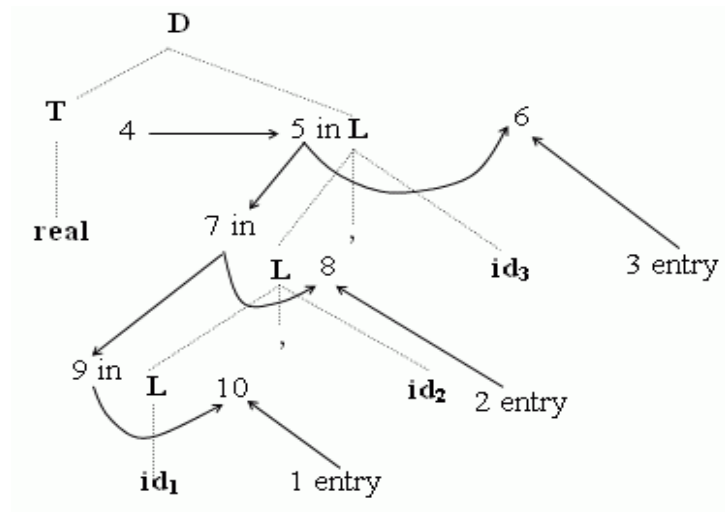
**Ví dụ 5:** Với định nghĩa S- thuộc tính

$$E \rightarrow E_1 + E_2 \quad E.val := E_1.val + E_2.val$$

Ta có đồ thị phụ thuộc:



**Ví dụ 6:** Đồ thị phụ thuộc cho cây chú thích trong ví dụ 4

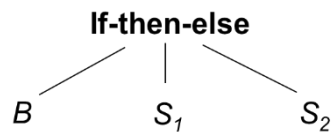


### 5.2.2. Xây dựng cây cú pháp

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:



Trong cây cú pháp các toán hạng và từ khoá không xuất hiện ở nút lá  
 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 mỗi toán hạng và toán tử
- Mỗi một nút có thể cài đặt bằng một bản ghi có nhiều trường
- Trong nút toán tử, có một trường chỉ toán tử, các trường còn lại chứa con trỏ, trỏ tới các nút toán hạng

Để xây dựng cây cú pháp cho biểu thức chúng ta sử dụng các hàm sau đây:

1. *mknnode*(*op*, *left*, *right*): Tạo một nút toán tử có nhãn là *op* và hai trường chứa con trỏ, trỏ tới *left* và *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 7:** Xây dựng cây cú pháp cho biểu thức:

**a - 4 + c** ta dùng một dãy các lời gọi các hàm

(1):  $p_1 := \text{mkleaf}(\mathbf{id}, \text{entry}_a)$       (4):  $p_4 := \text{mkleaf}(\mathbf{id}, \text{entry}_c)$

(2):  $p_2 := \text{mkleaf}(\mathbf{num}, 4)$       (5):  $p_5 := \text{mknnode}(' + ', p_3, p_4)$

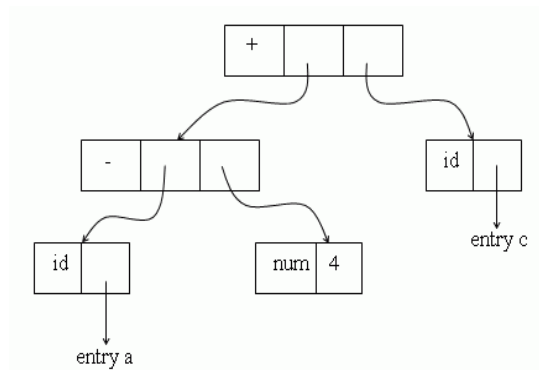
(3):  $p_3 := \text{mknnode}(' - ', p_1, p_2)$

**Cây được xây dựng từ dưới lên**

$p_1, p_2, \dots, p_5$  là các con trỏ, trỏ tới các nút

$\text{entry}_a, \text{entry}_c$  là các con trỏ, trỏ tới mục vào của a và c trong symbol table

Cây cú pháp cho biểu thức **a - 4 + c**

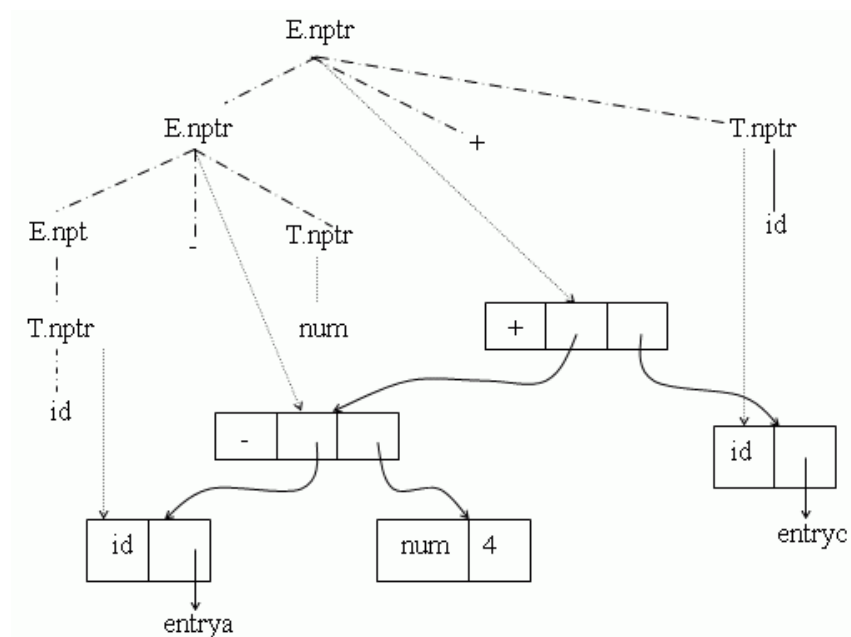


Xây dựng cây cú pháp từ định nghĩa trực tiếp cú pháp: Căn cứ vào các luật sinh văn phạm và luật ngữ nghĩa kết hợp mà ta gọi các hàm *mknnode* và *mkleaf* để tạo ra cây cú pháp

**Ví dụ 8:** Định nghĩa trực tiếp cú pháp giúp việc xây dựng cây cú pháp cho biểu thức (thuộc tính tổng hợp *nptr* theo dõi con trỏ được trả về khi gọi các hàm)

PRODUCTION	SYMANTIC RULES
$E \rightarrow E_1 + T$	$E.nptr := mknnode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknnode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow id$	$T.nptr := mkleaf(id, id.entry)$
$T \rightarrow num$	$T.nptr := mkleaf(num, num.val)$

**Ví dụ 9:** Cây cú pháp cho biểu thức **a-4+c**



Đánh giá từ dưới lên với định nghĩa S- thuộc tính:

- Các thuộc tính tổng hợp được đánh giá bằng cách phân tích cú pháp từ dưới lên
- Bộ phân tích cú pháp lưu trữ giá trị các thuộc tính và các kí hiệu văn phạm trong STACK
- Khi áp dụng reduction, giá trị tổng hợp mới được tạo từ các thuộc tính của các kí tự văn phạm bên vế phải luật sinh được lưu trữ trong STACK
- STACK được cài đặt bởi một cặp mảng state và val. 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
- 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] (vị trí của X trước đó) và thuộc tính tổng hợp nằm trong val[top].

	state	Val
	...	...
	X	X.x
	Y	Y.y
top→	Z	Z.z
	...	...

**Ví dụ 10:** Biểu diễn một máy tính đơn giản với LR parser

PRODUCTION	CODE FRAGMENT
$L \rightarrow En$	print(val[top])
$E \rightarrow E_1 + T$	val[ntop] := val[top - 2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[ntop] := val[top - 2] * val[top]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntop] := val[top - 1]
$F \rightarrow \text{digit}$	

Bảng sau trình bày quá trình thực hiện của bộ phân tích cú pháp với chuỗi nhập vào **3\*5+4 n**

INPUT	STATE	VAL	PRODUCTION USED
3 * 5 + 4 n	—	—	
* 5 + 4 n	3	3	
* 5 + 4 n	F	3	F → digit
*5 + 4 n	T	3	T → F
5 + 4 n	T*	3 -	
+ 4 n	T * 5	3 - 5	
+ 4 n	T * F	3 - 5	F → digit
+ 4 n	T	15	T → T * F
+ 4 n	E	15	E → T
4 n	E +	15 -	
n	E + 4	15 - 4	
n	E + F	15 - 4	F → digit
n	E + T	15 - 4	T → F
n	E	19	E → E + T
	E n	19 -	
	L	19	L → En

### 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  $X_j$  ( $1 \leq j \leq n$ ) trong vế phải của luật sinh  $A \rightarrow X_1X_2...X_n$  chỉ phụ thuộc vào:

1. Các thuộc tính của  $X_1, X_2, ..., X_{j-1}$
2. Các thuộc tính kế thừa của A.

### 5.2.3. Lược đồ dịch

Lược đồ dịch (translation scheme) 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 đặt trong cặp dấu  $\{ \}$  được xen vào bên phải của luật sinh

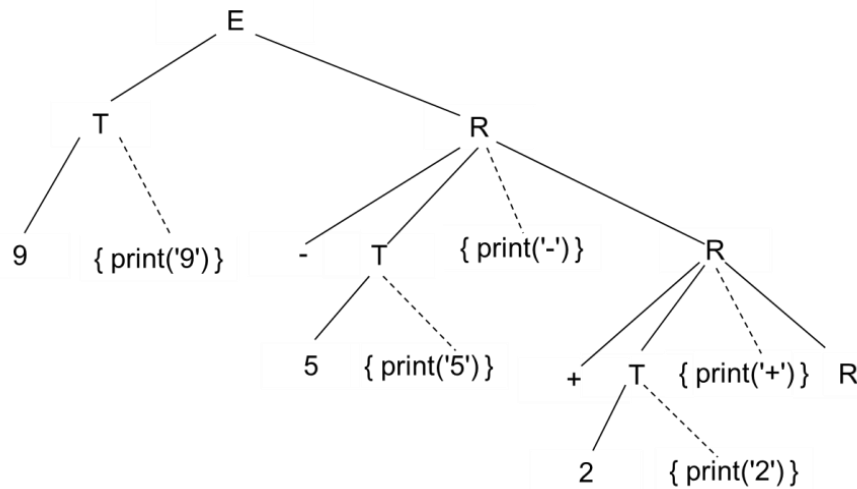
**Ví dụ 11:** Lược đồ dịch một biểu thức trung tố với phép cộng và trừ thành dạng hậu tố:

$E \rightarrow T R$

$R \rightarrow \text{addop } T \{ \text{print} ( \text{addop.lexeme} ) \} R_1 \mid \epsilon$

$T \rightarrow \text{num} \quad \{ \text{print}(\text{num.val}) \}$

Với biểu thức  $9 - 5 + 2$  ta có cây phân tích cú pháp phía dưới. Khi áp dụng phép duyệt cây depth- first sẽ có kết quả dạng hậu tố là  $95-2+$



Cách xây dựng một lược đồ dịch

Trường hợp chỉ chứa thuộc tính tổng hợp: Với mỗi một luật ngữ nghĩa, ta tạo ra lệnh gán tương ứng và đặt vào cuối vế phải luật sinh

Trường hợp chứa cả thuộc tính tổng hợp và thuộc tính kế thừa phải thỏa mãn 3 điều kiện:

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

#### 5.2.4. 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 để 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.

### **- Nội dung thảo luận**

So sánh sự giống và khác nhau giữa quá trình phân tích ngữ nghĩa của ngôn ngữ tự nhiên với phân tích ngữ nghĩa trong trình biên dịch.

### **- Yêu cầu SV chuẩn bị**

Xây dựng lược đồ dịch cho các ví dụ trong bài giảng.

Đánh giá dưới lên đối với định nghĩa S\_thuộc tính.

Vấn đề loại bỏ đệ quy trái trong xây dựng lược đồ dịch.

### **- Bài tập bắt buộc**

1. Văn phạm sau dùng để xây dựng các biểu thức sử dụng phép + giữa hằng nguyên và thực. Cộng 2 số nguyên thì kết quả trả về là một số nguyên, các trường hợp còn lại trả về số thực

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \text{num.num} \mid \text{num}$$

Hãy đưa ra định nghĩa dịch dựa cú pháp để xác định kiểu của mỗi biểu thức con. Chuyển biểu thức đó về dạng ký pháp hậu tố ngay trong khi xác định kiểu. Sử dụng phép toán inttoreal để chuyển đổi 1 giá trị nguyên thành 1 giá trị thực tương ứng để thực hiện phép + với các toán hạng cùng kiểu.

2. Sau đây là văn phạm dùng miêu tả các khai báo đầu chương trình:

$$D \rightarrow \text{id } L$$
$$L \rightarrow , \text{id } L \mid :T$$
$$T \rightarrow \text{integer} \mid \text{real}$$

Hãy dựng lược đồ chuyển đổi để nhập kiểu của các tên vào bảng ký hiệu.

3. Xây dựng một cây phân tích cú pháp chú thích cho biểu thức số học sau:

$$(4 * 7 + 1) * 2$$

4. 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. Xây dựng một DAG cho các biểu thức sau đây:

- a)  $a + a + (a + a + a + (a + a + a + a))$
- b)  $x * (3 * x + x * x)$

### - Bài tập nâng cao

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.

7. Chọn một phương pháp gán dịch dựa cú pháp với các luật cú pháp và viết chương trình đánh giá chúng.

### - Tài liệu tham khảo

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 5.
2. **Engineering a Compiler.** K. Cooper, L. Torczon. - Morgan-Kaufman Publishers, 2005. Chương 5.
3. **Giáo trình chương trình dịch** 2<sup>nd</sup> Edition. Phạm Hồng Nguyên. NXB ĐHQG Hà Nội, 2009. Chương 5.

### - Câu hỏi ôn tập

1. Thế nào là biên dịch dựa cú pháp?
2. Đồ thị phụ thuộc là gì?
3. Lược đồ dịch là gì, cách xây dựng lược đồ dịch?

## Bài giảng 06: Phân tích ngữ nghĩa – Kiểm tra kiểu

Chương 6, mục:

Tiết thứ: 34-36

Tuần thứ: 12

**- Mục đích yêu cầu**

**Mục đích:** Nắm được cách định nghĩa hệ thống kiểu trong các ngôn ngữ lập trình; Cách kiểm tra kiểu trong quá trình biên dịch. 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.

**Yêu cầu:** 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).

**- Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

**- Thời gian:** Giáo viên giảng: 2 tiết; Thảo luận và làm bài tập trên lớp: 1 tiết; Sinh viên tự học: 6 tiết.

**- Địa điểm:** Giảng đường do P2 phân công.

**- Nội dung chính:**

5.1. Các vấn đề về kiểm tra kiểu

5.1.1. Nhiệm vụ

5.1.2. Kiểm tra kiểu

5.2. Sự tương đương của các biểu thức kiểu

5.3. Chuyển đổi kiểu

5.4. Vấn đề xử lý lỗi ngữ nghĩa

**5.1.1. Nhiệm vụ**

Làm nhiệm vụ kiểm tra tính đúng đắn về mặt ngữ nghĩa (phát hiện tất cả các lỗi không phải lỗi cú pháp) của chương trình nguồn, việc kiểm tra chia làm 2 loại: kiểm tra tĩnh và kiểm tra động (xảy ra lúc chương trình đang thực thi). Ta chỉ xét một số dạng kiểm tra tĩnh:

- Kiểm tra kiểu: tính đúng đắn về kiểu của các toán hạng trong biểu thức.
- Kiểm tra dòng điều khiển: ví dụ lệnh break trong C.
- Kiểm tra tính nhất quán: ví dụ trong khai báo các biến, lệnh case...
- Kiểm tra quan hệ tên: tên có thể phải xuất hiện từ 2 lần trở lên.

Kiểm tra kiểu được thực hiện dựa trên cây cú pháp, tạo ra các thông tin cần thiết để sinh mã

**5.1.2. Kiểm tra kiểu**



Rất khác nhau đối với các ngôn ngữ khác nhau. Các thông tin về kiểu (có được sau khi kiểm tra kiểu) có thể cần thiết để sinh mã

### **Biểu thức kiểu**

Kiểu của một ngôn ngữ lập trình được kí hiệu bởi các biểu thức kiểu (type expression).

Biểu thức kiểu được định nghĩa như sau:

1. Kiểu cơ sở là một biểu thức kiểu: `boolean`, `char`, `integer`, `real`, `type_error`, `void`
2. Một tên kiểu là một biểu thức kiểu
3. Mỗi kiểu dữ liệu có cấu trúc 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 (product): Nếu  $T_1, T_2$  là biểu thức kiểu thì tích Đề- các  $T_1 * T_2$  là biểu thức kiểu
  - c. Bản ghi (record): Là cấu trúc bao gồm một bộ các tên trường, kiểu trường
  - d. Con trỏ (pointer): Nếu  $T$  là một biểu thức kiểu thì  $\text{pointer}(T)$  là một biểu thức kiểu  $T$
  - e. Hàm (function): Hàm là một ánh xạ các phần tử của tập xác định (domain)  $D$  lên tập giá trị (range)  $R$ . Một hàm là một biểu thức kiểu  $D \rightarrow R$

### **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 định danh được khai báo trước khi sử dụng

Bộ kiểm tra kiểu (type checker) là một lược đồ dịch, nó tổng hợp kiểu của mỗi biểu thức từ kiểu của các biểu thức con của nó

Định nghĩa 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  $E$

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid \text{id} : T$

$T \rightarrow \text{char} \mid \text{integer} \mid \text{array}[\text{num}] \text{ of } T \mid \uparrow T$

$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E [E] \mid E \uparrow$

**Ví dụ 1:** Chương trình sau sinh bởi văn phạm trên

key: integer;

key mod 1999

Ta có lược đồ dịch để lưu trữ kiểu của một định danh

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T \quad \{ \text{addtype}(\text{id.entry}, T.\text{type}) \}$

$T \rightarrow \text{char} \quad \{ T.\text{type} := \text{char} \}$

$T \rightarrow \text{integer} \quad \{ T.\text{type} := \text{integer} \}$

$T \rightarrow \uparrow T_1 \quad \{ T.\text{type} := \text{pointer}(T_1.\text{type}) \}$

$T \rightarrow \text{array}[\text{num}] \text{ of } T_1 \quad \{ T.\text{type} := \text{array}(1 \dots \text{num.val}, T_1.\text{type}) \}$

### Kiểm tra kiểu của các biểu thức

Lược đồ dịch cho kiểm tra kiểu của biểu thức:

$E \rightarrow \text{literal} \quad \{ E.\text{type} := \text{char} \}$

$E \rightarrow \text{num} \quad \{ E.\text{type} := \text{integer} \}$

$E \rightarrow \text{id} \quad \{ E.\text{type} := \text{lookup}(\text{id.entry}) \}$

$E \rightarrow E_1 \text{ mod } E_2 \quad \{ E.\text{type} := \text{if } E_1.\text{type} = \text{integer and } E_2.\text{type} = \text{integer}$

$\text{then integer else type\_error} \}$

$E \rightarrow E_1[E_2] \quad \{ E.\text{type} := \text{if } E_2.\text{type} = \text{integer and } E_1.\text{type} = \text{array}(s,t)$

$\text{then } t \text{ else type\_error} \}$

$E \rightarrow E_1 \uparrow \quad \{ E.\text{type} := \text{if } E_1.\text{type} = \text{pointer}(t) \text{ then } t \text{ else type\_error} \}$

### Kiểm tra kiểu của các câu lệnh

Các câu lệnh cấu tạo lên ngôn ngữ không có giá trị, do đó ta gán cho chúng kiểu *void*

Lược đồ dịch cho kiểm tra kiểu của các lệnh:

$S \rightarrow \text{id} := E \quad \{ S.\text{type} := \text{if } \text{id.type} = E.\text{type} \text{ then void else type\_error} \}$

$S \rightarrow \text{if } E \text{ then } S_1 \quad \{ S.\text{type} := \text{if } E.\text{type} = \text{boolean then } S_1.\text{type} \text{ else type\_error} \}$

$S \rightarrow \text{while } E \text{ do } S_1 \quad \{ S.\text{type} := \text{if } E.\text{type} = \text{boolean then } S_1.\text{type} \text{ else type\_error} \}$

$S \rightarrow S_1 ; S_2 \quad \{ S.\text{type} := \text{if } S_1.\text{type} = \text{void and } S_2.\text{type} = \text{void then void}$

$\text{else type\_error} \}$

### Kiểm tra kiểu của các hàm

Việc ghép một hàm với một đối (argument) có thể diễn đạt bởi luật sinh:  $E \rightarrow E(E)$

Lược đồ dịch kiểm tra kiểu cho một hàm:

$$E \rightarrow E_1(E_2) \quad \{ E.type := \text{if } E_2.type = s \\ \text{and } E_1.type = s \rightarrow t \text{ then } t \\ \text{else type\_error} \}$$

Nếu có nhiều đối có kiểu tương ứng  $T_1, T_2, \dots, T_n$  được coi như một đối duy nhất có kiểu  $T_1 * T_2 * \dots * T_n$

## 5.2. Sự tương đương của các biểu thức kiểu

**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:

- 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.

**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;
```

## 5.3. Chuyển đổi kiểu

Xét biểu thức  $x + i$  trong đó  $x$  có kiểu real và  $i$  có kiểu integer. 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 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 intto real real+** (**intto real** đổi số nguyên  $i$  thành số thực, **real+** thực hiện phép cộng các số thực)

**Ép kiểu (coercion):** Việc chuyển một kiểu dữ liệu sang một kiểu khác được gọi là ẩn (implicit) nếu nó được làm tự động bởi compiler và được gọi là hiện (explicit) nếu được giải quyết bởi người lập trình

**Ví dụ 2:** Hàm ord() trong pascal chuyển kiểu kí tự sang số nguyên, phép chuyển đổi là hiện

Lệnh a:=b; trong đó a kiểu real, b kiểu integer khi thực hiện compiler sẽ thực hiện ép kiểu biến b sang kiểu real trước khi thực hiện lệnh gán, phép chuyển đổi là ẩn.

### 5.5. Vấn đề xử lý lỗi ngữ nghĩa

- Thông báo về lỗi
- Mô tả và vị trí lỗi
- Khắc phục lỗi để tiếp tục kiểm tra phần chương trình nguồn còn lại
- Lỗi xuất hiện gây ảnh hưởng đến các luật kiểm tra lỗi: phải thiết kế kiểu hệ thống để các luật có thể đương đầu với các lỗi kiểu

#### - Nội dung thảo luận

Các lỗi ngữ nghĩa hay gặp khi lập trình trên Visual C++.

Bộ đặc tả kiểm tra kiểu của ngôn ngữ C++ trong môi trường Turbo C++.

#### - Yêu cầu SV chuẩn bị

Các vấn đề khác liên quan đến kiểm tra kiểu, đọc chương 5 tài liệu 1, chương 5 tài liệu 2, chương 4 tài liệu 3.

#### - Bài tập bắt buộc

1. Xác định kiểu cho các câu lệnh sau:

- a) var a : integer; a := a + 0.5;
- b) var a : integer; if a+5 then a := 0;
- c) var a : integer; a := 0; while a<10 do a := a[0] - 1;

2. Văn phạm sau dùng để định danh danh sách ký hiệu là list:

- $P \rightarrow D; E$
- $D \rightarrow D; D \mid id : T$
- $T \rightarrow char \mid integer \mid list\ of\ T$
- $E \rightarrow literal \mid num \mid id \mid (L)$
- $L \rightarrow E, L \mid E$

Hãy viết các luật ngữ nghĩa để xác định kiểu cho E và L.

3. 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 các đối tượng là các số thực;
- b) Mảng hai chiều của các số nguyên, hàng có kích thước từ 0 đến 9, cột 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 các ký tự.

**- Bài tập nâng cao**

4. 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.

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 miền giá trị mà nó sẽ nằm trong đó.

6. 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.

7. Cài đặt khối phân tích ngữ nghĩa.

8. Thiết kế và cài đặt bảng ký hiệu cho chương trình dịch.

**- Tài liệu tham khảo**

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 5.
2. **Giáo trình chương trình dịch** 2<sup>nd</sup> Edition. Phạm Hồng Nguyên. NXB ĐHQG Hà Nội, 2009. Chương 5.

**- Câu hỏi ôn tập**

1. Mục đích, nhiệm vụ của kiểm tra kiểu.
2. Sự tương đương giữa các biểu thức kiểu.
3. Ép kiểu là gì, các trường hợp ép kiểu. Cho ví dụ minh họa.

**Bài giảng 07: Sinh mã trung gian**

Chương 7, mục:

Tiết thứ: 37-39

Tuần thứ: 13

**- Mục đích yêu cầu**

**Mục đích:** 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.

**Yêu cầu:** sinh viên phải hệ thống lại các kiến thức cơ sở về toán rời rạc, kiến thức lập trình, tự nghiên cứu và ôn tập lại những vấn đề lý thuyết khác có liên quan đến môn học.

**- Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

**- Thời gian:** Giáo viên giảng: 2 tiết; Thảo luận và làm bài tập trên lớp: 1 tiết; Sinh viên tự học: 6 tiết.

**- Địa điểm:** Giảng đường do P2 phân công.

**- Nội dung chính:**

6.1. Ngôn ngữ trung gian, mã 3 địa chỉ

6.2. Sinh mã trung gian cho biểu thức số học

6.3. Sinh mã trung gian cho biểu thức boole, logic và số học

6.4. Sinh mã trung gian cho một số lệnh điều khiển

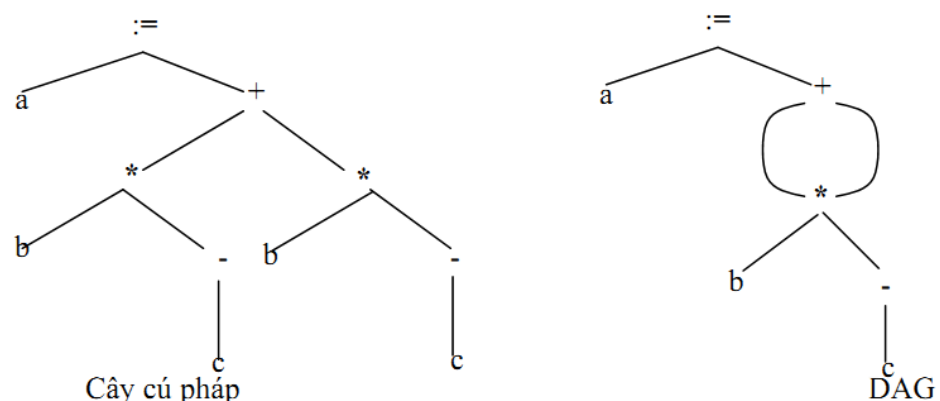
**6.1. Ngôn ngữ trung gian**

Cây cú pháp, ký pháp hậu tố và mã 3 địa chỉ là các loại biểu diễn trung gian.

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 1: Với lệnh gán  $a := b * -c + b * -c$ , ta có cây cú pháp và DAG:



### 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,  $\text{op}$  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.

### Các mã lệnh 3 địa chỉ phổ biến:

1. Lệnh gán dạng  $x := y \text{ op } z$ , trong đó  $\text{op}$  là toán tử 2 ngôi số học hoặc logic.

2. Lệnh gán dạng  $x := \text{op } y$ , trong đó  $\text{op}$  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  $\text{if } x \text{ relop } y \text{ goto L}$ . Lệnh này áp dụng toán tử quan hệ relop ( $<, =, >, \dots$ ) 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  $x \text{ relop } y \text{ goto L}$  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  $p (x1, x2, \dots, xn)$ .

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  $y$ .

Ví dụ 2: Định nghĩa S-thuộc tính sinh mã lệnh địa chỉ cho lệnh gán:

Luật sinh	Luật ngữ nghĩa
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E1 + E2$	$E.place := newtemp;$ $E.code := E1.code \parallel E2.code \parallel$ $gen(E.place := E1.place '+' E2.place)$
$E \rightarrow E1 * E2$	$E.place := newtemp;$ $E.code := E1.code \parallel E2.code \parallel$ $gen(E.place := E1.place '*' E2.place)$
$E \rightarrow - E1$	$E.place := newtemp;$ $E.code := E1.code \parallel gen(E.place := 'uminus' E1.place)$
$E \rightarrow (E1)$	$E.place := newtemp;$ $E.code := E1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

Trong đó 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  $E$ .

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
```

## 6.2. Sinh mã trung gian cho biểu thức số học

Sản xuất	Luật ngữ nghĩa
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E1 + E2$	$E.place := newtemp;$ $E.code := E1.code \parallel E2.code \parallel gen(E.place := E1.place '+' E2.place)$
$E \rightarrow E1 * E2$	$E.place := newtemp;$ $E.code := E1.code \parallel E2.code \parallel gen(E.place := E1.place '+' E2.place)$



E -> - E1	E.place := newtemp; E.code := E1.code    gen(E.place ':=' 'uminus' E1.place)
E -> ( E1 )	E.place := E1.place E.code := E1.code
E -> id	E.place := id.place E.code := ''

### 6.3. Sinh mã trung gian cho biểu thức boole, logic và số học

#### Sinh mã đích cho biểu thức boole

Đối với một biểu thức Boole E, ta dịch E thành một dãy các câu lệnh ba địa chỉ, trong đó đối với các phép toán logic sẽ sinh ra các lệnh nhảy có điều kiện và không có điều kiện đến một trong hai vị trí: E.true, nơi quyền điều khiển sẽ chuyển tới nếu E đúng, và E.false, nơi quyền điều khiển sẽ chuyển tới nếu E sai.

#### Sinh mã trung gian cho biểu thức logic và số học

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

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.

### 6.4. Sinh mã trung gian cho một số lệnh điều khiển

Để sinh ra một nhãn mới, ta dùng thủ tục newlabel. 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.

\* Dịch biểu thức logic trong các lệnh điều khiển

Nếu E có dạng  $a < b$  thì mã lệnh sinh ra có dạng

```
if a < b goto E.true
goto E.false
```

Nếu E có dạng E1 or E2. Nếu E1 là true thì E là true. Nếu E1 là false thì phải đánh giá E2. Do đó E1.false là nhãn của lệnh đầu tiên của E2. E sẽ true hay false phụ thuộc vào E2 là true hay false.

Tương tự cho E1 and E2.

Nếu E có dạng not E1 thì E1 là true thì E là false và ngược lại.

**- Nội dung thảo luận**

Sinh mã đích cho các biểu thức điều kiện và vòng lặp, switch...case...

**- Yêu cầu SV chuẩn bị**

Tìm hiểu về sinh mã đích cho các cấu trúc khác trong ngôn ngữ lập trình: khai báo, hàm...

Tìm hiểu về MSIL và bộ sinh mã trung gian trong môi trường VS Studio.net.

**- Bài tập bắt buộc**

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.
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.
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)  $x = a / ( b + c ) - d * ( e + f )$

**- Bài tập nâng cao**

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)  $a[i] += b[j]$
5. Dịch lệnh gán sau thành mã máy 3 - địa chỉ:

$$A[i, j] := B[i, j] + C[A[k, l]] + D[i + j]$$

6. Chuyển đoạn chương trình C sau thành mã ba địa chỉ:

```
int i;
int a[10]; i = 1;
while (i <= 10) { a[i] = 0; i = i + 1; }
```

#### **- Tài liệu tham khảo**

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 6.
2. **Advanced Compiler Design and Implementation.** S. Muchnick. Morgan-Kaufmann Publishers, 2007. Chương 6.

#### **- Câu hỏi ôn tập**

1. Mã ba địa chỉ là gì? Nêu các dạng mã ba địa chỉ phổ biến.
2. Sinh mã trung gian cho biểu thức số học
3. Sinh mã trung gian cho biểu thức boole, logic và số học
4. Sinh mã trung gian cho một số lệnh điều khiển

### **Bài giảng 08: Sinh mã đích**

Chương 8, mục:

Tiết thứ: 40-42

Tuần thứ: 14

#### **- Mục đích yêu cầu**

**Mục đích:** 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.

**Yêu cầu:** sinh viên phải hệ thống lại các kiến thức kiến thức về kiến trúc máy tính đặc biệt là phân hợp ngữ (assembly language) để thuận tiện cho việc tiếp nhận kiến thức về máy đích.

**- Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

**- Thời gian:** Giáo viên giảng: 2 tiết; Thảo luận và làm bài tập trên lớp: 1 tiết; Sinh viên tự học: 6 tiết.

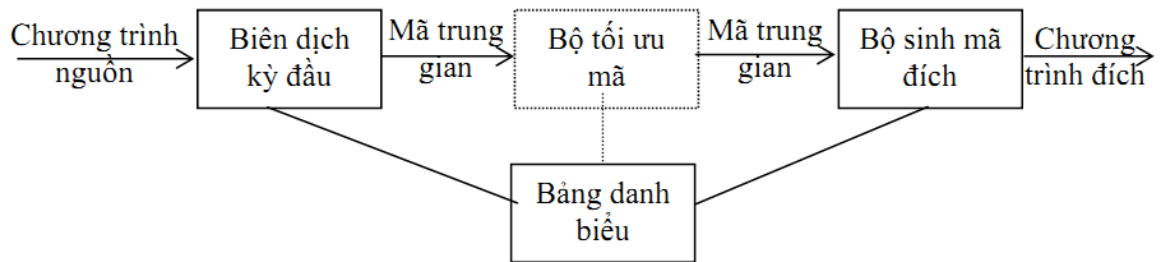
**- Địa điểm:** Giảng đường do P2 phân công.

#### **- Nội dung chính:**

7.1. Mục tiêu, đầu vào đầu ra của code generation

- 7.2. Các dạng mã máy đối tượng
- 7.3. Các vấn đề thiết kế bộ sinh mã
- 7.4. Sinh mã đích cho biểu thức số học
- 7.5. Sinh mã đích cho biểu thức boole, logic và số học
- 7.6. Sinh mã ba địa chỉ cho một số lệnh điều khiển
- 7.7. Ví dụ về bộ sinh mã đơn giản

### 7.1. Mục tiêu, đầu vào đầu ra của code generation



Đầu vào:

- Chương trình trong ngôn ngữ trung gian
- Bảng ký hiệu

Đầu ra có nhiều dạng:

- Một chương trình hoàn toàn bằng ngôn ngữ máy, chạy được ngay
- Một chương trình bằng ngôn ngữ máy có thể định vị lại được (relocatable)
- Một chương trình bằng ngôn ngữ Assembly
- Một chương trình bằng một ngôn ngữ lập trình khác

Sinh mã tốt khó:

- Mã thường gắn với máy tính cụ thể
- Tốc độ của chương trình đích có thể rất khác nhau tùy thuộc thuật toán sinh mã
- Sinh mã đúng là tiêu chí quan trọng nhất
- Để sinh mã tốt, cần nhiều nghiên cứu thực hành

### 7.2. Các dạng mã máy đối tượng

#### Mã máy định vị tuyệt đối

Các lệnh mã máy được đặt ở các vị trí tuyệt đối

Chương trình dịch tạo ra chương trình đối tượng hoàn chỉnh (không cần thêm bất cứ điều chỉnh nào khác)

Ưu: dịch nhanh, chạy nhanh

Nhược: phải dịch toàn bộ chương trình dù chỉ thay đổi một phần; mã máy đặt cố định trong bộ nhớ

### **Mã đối tượng định vị lại được**

Mã đối tượng lưu ở dạng các mô đun có thể định vị lại được: địa chỉ câu lệnh không được xác định

Linker: Liên kết một hoặc nhiều mô đun mã đối tượng thành một chương trình

Loader: biến đổi thành chương trình đối tượng tuyệt đối

Ưu điểm:

Các mô đun nguồn độc lập, có thể dịch đi dịch lại độc lập với nhau

Dễ kết hợp với các thư viện

Thư viện và chương trình đối tượng có thể được viết và dịch trong ngôn ngữ lập trình khác

Nhược:

Quá trình dịch - liên kết - nạp tốn thời gian và chậm hơn sinh mã tuyệt đối

### **Mã đối tượng thông dịch**

Các lệnh dịch ra không phải là các lệnh máy hoạt động trực tiếp

Là các câu lệnh thông dịch trừu tượng

Chương trình đối tượng chạy nhờ một chương trình thông dịch:

- đọc vào từng mã lệnh
- giải mã từng lệnh
- gọi chạy một thủ tục tương ứng

Ưu điểm:

- Dễ viết chương trình dịch
- Chương trình đối tượng nhỏ gọn
- Các phương tiện đối thoại và gỡ rối trong khi chạy có thể được thêm vào bộ thông dịch mà không cần phải thêm vào bản thân mã đối tượng
- Tính khả chuyển cao

Nhược điểm: Chạy chậm

## **7.3. Các vấn đề thiết kế bộ sinh mã**

**Quản lý bộ nhớ:** Việc chuyển đổi tương ứng từ tên (biến, hàm...) trong chương trình nguồn thành địa chỉ dữ liệu trong lúc chạy của chương trình đích được thực hiện phối hợp giữa các phần trước và phần sinh mã.

**Sử dụng thanh ghi:** Các câu lệnh tính toán dùng thanh ghi thì thường nhanh và gọn hơn với bộ nhớ

Cần nhắc khi sử dụng thanh ghi:

Chọn tập các biến sẽ được đặt trong các thanh ghi

Chọn thanh ghi xác định để lưu một biến.

### **Thứ tự làm việc:**

Thứ tự tính toán có thể ảnh hưởng đến hiệu quả của mã đích

Một số thứ tự tính toán cần các nhiều thanh ghi giữ kết quả trung gian hơn một số thứ tự khác

Xác định được thứ tự tốt nhất là công việc rất khó.

### **Cấu tạo máy đích:**

Có một bộ nhớ được đánh địa chỉ theo từng byte

Có  $n$  thanh ghi đa năng  $R_0, R_1, \dots, R_{n-1}$

Các câu lệnh có hai địa chỉ dạng:

*op nguồn, đích*

Ví dụ một số mã phép toán:

MOV (chuyển nguồn vào đích)

ADD (cộng nguồn vào đích)

SUB (trừ nguồn với đích)

## **7.7. Ví dụ về bộ sinh mã đơn giản**

- Một bộ sinh mã đơn giản tạo ra mã máy từ mã trung gian ba địa chỉ.
- Mỗi toán tử trong các câu lệnh sẽ có một toán tử tương ứng trong mã máy
- Kết quả tính toán có thể đặt được trên thanh ghi
- Có thể tạo ra mã tốt hơn cho câu lệnh ba địa chỉ  $a := b + c$  nếu ta chỉ tạo ra một mã lệnh đơn ADD  $R_j, R_i$  và đặt kết quả  $a$  trong thanh ghi  $R_i$

Bộ diễn tả thanh ghi:

- Theo dõi nội dung các thanh ghi
- Được dùng mỗi khi cần một thanh ghi mới.

Bộ diễn tả địa chỉ:

- Theo dõi việc cấp phát giá trị hiện tại của tên gặp trong lúc chạy.
- Thông tin có thể được lưu trong bảng ký hiệu và được dùng để xác định phương pháp truy nhập đối với một tên.

Thuật toán sinh mã:

- Gọi hàm *gettreg()* để xác định vị trí  $L$  nơi phép tính  $a \text{ op } z$  thực hiện
- Gọi bộ mô tả địa chỉ cho  $y$  để xác định  $y'$  là vị trí hiện thời của  $y$ .  
Chọn  $y'$  là thanh ghi nếu giá trị  $y$  hiện tại vừa trong thanh ghi vừa

trong bộ nhớ. Nếu giá trị  $y$  không ở  $L$  thì sinh ra câu lệnh MOV  $y'$ ,  $L$  để đặt một bản sao của  $y$  vào  $L$ .

- Sinh ra lệnh OP  $z'$ ,  $L$  với  $z'$  là vị trí hiện thời của  $z$ . Chọn thanh ghi nếu  $z$  nằm ở cả thanh ghi lẫn bộ nhớ. Cập nhật lại địa chỉ của  $x$  để chỉ rằng  $x$  nằm tại vị trí  $L$ . Nếu  $L$  là một thanh ghi, cập nhật bộ mô tả của nó để chỉ rằng nó chứa giá trị của  $x$  và loại  $x$  khỏi các bộ mô tả thanh ghi khác.
- Nếu giá trị hiện thời của  $y$  hoặc  $z$  (hoặc cả hai) không được dùng tiếp nữa, không tồn tại khi kết thúc khối và đang nằm trên các thanh ghi, thì chọn bộ mô tả thanh ghi để chỉ rằng sau khi thực hiện  $x := y \text{ op } z$  thì các thanh ghi đó sẽ không còn chứa  $y, z$  nữa.

Hàm Getreg:

1. Nếu tên  $y$  có trong một thanh ghi và  $y$  không được dùng tiếp sau khi thực hiện  $x := y \text{ op } z$ , thì trả lại  $y$  cho  $L$ . Cập nhật bộ mô tả địa chỉ cho  $y$  để chỉ rằng  $y$  không còn tồn tại trong  $L$  nữa.
2. Quay lại 1, trả lại một thanh ghi rỗng cho  $L$  nếu có một.
3. Quay lại 2 nếu  $x$  được dùng tiếp trong khối, hoặc  $\text{op}$  là một phép toán lấy chỉ số (đòi hỏi thanh ghi), tìm một thanh ghi dùng được  $R$ . Lưu giá trị  $R$  tại một nơi nào đó trong bộ nhớ (bằng lệnh MOV  $R, M$ ) nếu nó không sẵn trong vị trí bộ nhớ thích hợp  $M$  thì cập nhật bộ mô tả địa chỉ cho  $M$  và trả lại  $R$ . Nếu  $R$  giữ giá trị của vài biến, cần phải sinh ra một lệnh MOV cho mỗi giá trị cần lưu. Một thanh ghi thích hợp có thể là cái trỏ đến các dữ liệu xa nhất về sau này, hoặc cái mà dữ liệu của nó cũng có trong bộ nhớ.
4. Nếu  $x$  không được dùng trong khối, hoặc không tìm được thanh ghi thích hợp, thì chọn vị trí trong bộ nhớ của  $x$  cho  $L$ .

Câu lệnh	Mã được sinh	Bộ mô tả thanh ghi	Bộ mô tả địa chỉ
		Các thanh ghi rỗng	
$t := a - b$	MOV a, R0 SUB b, R0	R0 chứa t	t trong R0
$u := a - c$	MOV a, R1 SUB c, R1	R0 chứa t R1 chứa u	t trong R0 u trong R1
$v := t + u$	ADD R1, R0	R0 chứa v R1 chứa u	u trong R1 v trong R0
$d := v + u$	ADD R1, R0 MOV R0, d	R0 chứa d	d trong R0 d trong R0 và

			trong bộ nhớ
--	--	--	--------------

- **Nội dung thảo luận:** Sinh mã đích cho các biểu thức điều kiện và vòng lặp

- **Yêu cầu SV chuẩn bị**

Tìm hiểu về sinh mã đích cho các cấu trúc khác trong ngôn ngữ lập trình: khai báo, hàm...

- **Bài tập bắt buộc**

1. Chuyển các câu lệnh hoặc đoạn chương trình sau thành mã ba địa chỉ:

1)  $a * -(b+c)$

2) đoạn chương trình C

```
main ()
{ int i; int a[100];
  i=1;
  while(i<=10)
  { a[i]=0;
    i=i+1;}
}
```

2. 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.

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.

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)  $x = a / (b + c) - d * (e + f)$

- **Bài tập nâng cao**

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)  $a[i] += b[j]$

5. Dịch lệnh gán sau thành mã máy 3 - địa chỉ :



$$A[i,j] := B[i,j] + C[A[k,l]] + D[i+j]$$

**- Tài liệu tham khảo**

1. **Compilers : Principles, Technique and Tools.** A.V. Aho, M. Lam, R. Sethi, J.D.Ullman. - Addison -Wesley 2nd Edition, 2007. Chương 7.
2. **Advanced Compiler Design and Implementation.** S. Muchnick. Morgan-Kaufmann Publishers, 2007. Chương 6.
3. **Giáo trình chương trình dịch** 2<sup>nd</sup> Edition. Phạm Hồng Nguyên. NXB ĐHQG Hà Nội, 2009. Chương 6.

**Câu hỏi ôn tập**

1. Các dạng mã máy đối tượng.
2. Mã 3 địa chỉ là gì. Cho ví dụ minh họa.
3. Anh\chị hãy trình bày giải thuật sinh mã đích.

**Bài giảng 09: Các vấn đề của trình biên dịch hiện đại**

Chương 9, mục:

Tiết thứ: 43-45

Tuần thứ: 15

**- Mục đích yêu cầu**

**Mục đích:** Sau khi học xong bài 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.

**Yêu cầu:** Sinh viên phải nắm vững ít nhất một 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).

**- Hình thức tổ chức dạy học:** Lý thuyết, thảo luận, tự học, tự nghiên cứu

**- Thời gian:** Giáo viên giảng: 2 tiết; Thảo luận và làm bài tập trên lớp: 1 tiết; Sinh viên tự học: 6 tiết.

**- Địa điểm:** Giảng đường do P2 phân công.

**- Nội dung chính:**

- 8.1. Ngôn ngữ hướng đối tượng
- 8.2. Run-Time Environment
  - 8.2.1. Chương trình con và cây hoạt động
  - 8.2.2. Ngăn xếp điều khiển
  - 8.2.3. Tầm vực của sự khai báo
  - 8.2.4. Liên kết tên

- 8.2.5. Một số vấn đề cần quan tâm khi viết chương trình dịch
- 8.3. Bảng ký hiệu
- 8.4. Quản lý bộ nhớ và địa chỉ
- 8.5. Vấn đề tối ưu mã

## 8.1. Ngôn ngữ hướng đối tượng

## 8.2. Run-Time Environment

### 8.2.1. Chương trình con và cây hoạt động

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.

**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.

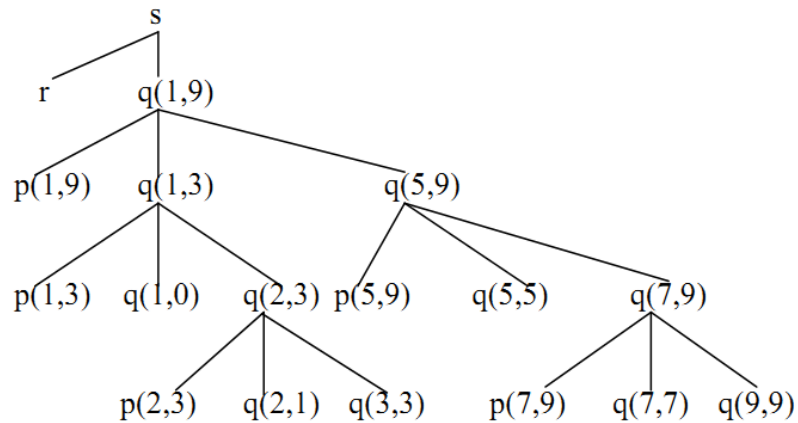
- Thời gian tồn tại của một chương trình con  $p$  là một chuỗi các bước giữa bước đầu tiên và bước cuối cùng trong sự thực hiện thân chương trình con bao gồm cả thời gian thực hiện các chương trình con được gọi bởi  $p$ .

- 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.

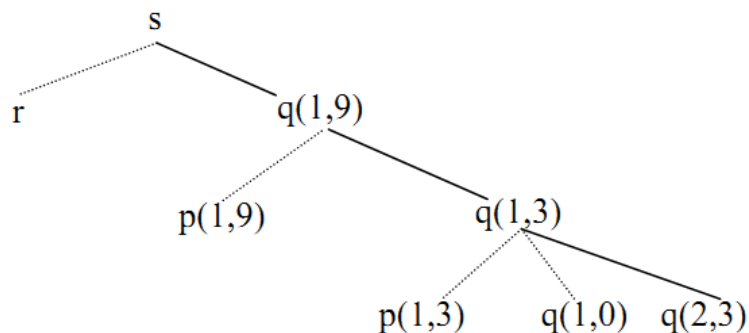
1. Mỗi nút biểu diễn cho một hoạt động của một chương trình con.
2. Nút gốc biểu diễn cho hoạt động của chương trình chính.
3. Nút  $a$  là cha của  $b$  nếu và chỉ nếu dòng điều khiển sự hoạt động đó từ  $a$  sang  $b$
4. Nút  $a$  ở bên trái của nút  $b$  nếu thời gian tồn tại của  $a$  xuất hiện trước thời gian tồn tại của  $b$ .



### 8.2.2. 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.

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.



### 8.2.3. 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).

### 8.2.4. 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.

### 8.2.5. Một số vấn đề cần quan tâm khi viết 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?

### 8.3. Bảng ký hiệu

Mục đích, nhiệm vụ: Thu thập các thông tin về tên; Sử dụng các thông tin về tên

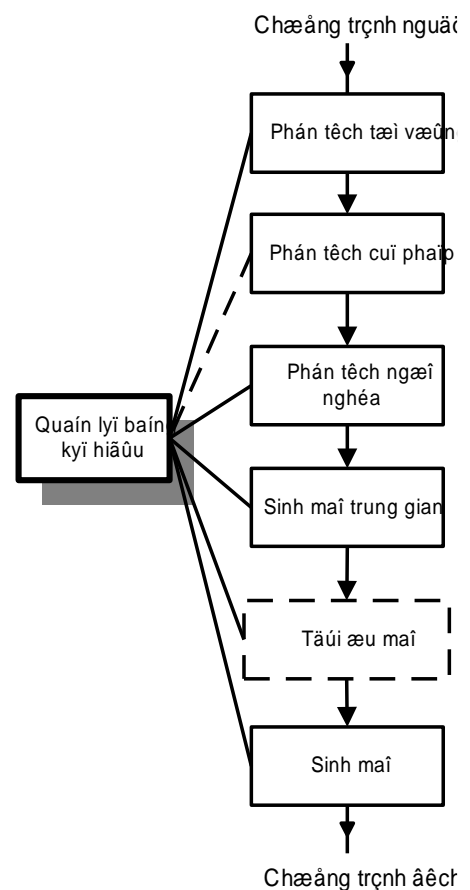
- ♦ Phân tích ngữ nghĩa: kiểm tra việc dùng các tên phải khớp với khai báo
- ♦ Sinh mã: lấy kích thước, loại bộ nhớ phải cấp phát cho một tên
- ♦ Các khối khác: để phát hiện và khắc phục lỗi

Các yêu cầu với bảng ký hiệu:

- ♦ phát hiện một tên cho trước có ở trong bảng hay không
- ♦ thêm một tên mới vào bảng
- ♦ lấy thông tin tương ứng với tên cho trước
- ♦ thêm các thông tin mới vào một tên cho trước
- ♦ xoá một tên hoặc nhóm tên trong bảng

Thông tin đưa vào bảng ký hiệu:

- ♦ Xâu ký tự tạo nên tên
- ♦ Thuộc tính của tên
- ♦ Các tham số, như số chiều của mảng
- ♦ (Có thể) con trỏ đến tên cấp phát



#### 8.4. Quản lý bộ nhớ và địa chỉ

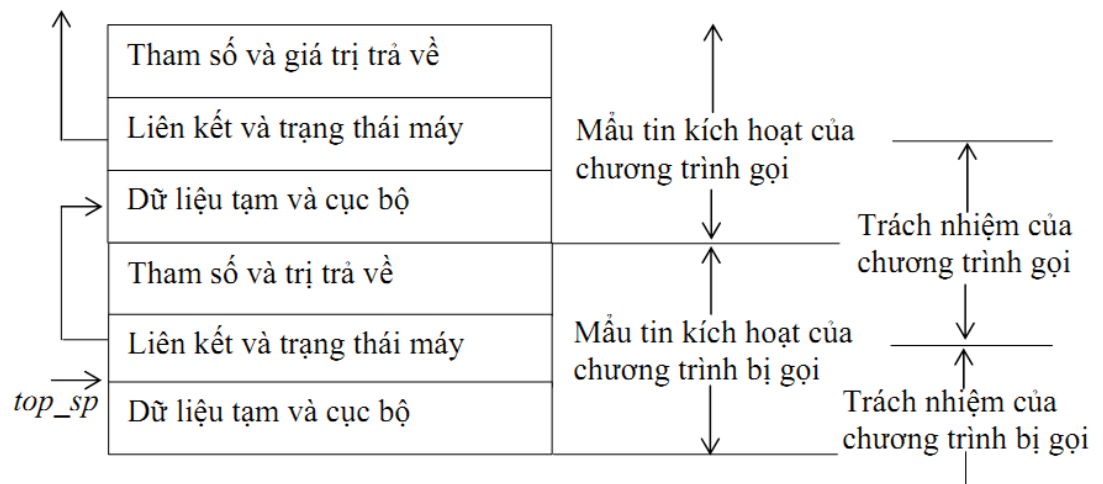
Bộ nhớ có thể chia ra để lưu trữ các phần:

1. Mã đích.
  2. Đối tượng dữ liệu.
  3. Bản sao của Stack điều khiển để lưu trữ hoạt động của chương trình con.
- 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.

Gọi thực hiện chương trình con:



#### 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.

- **Nội dung thảo luận:** vấn đề cấp phát bộ nhớ và địa chỉ

- **Nội dung tự học:** Tìm hiểu về các kỹ năng tối ưu mã chương trình, kỹ năng tối ưu mã nguồn.

- **Bài tập bắt buộc**

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.
```

2. 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:

$a = 3; \quad b = 4; \quad c = 4; \quad k = 2; \quad l = 3; \quad x = 4; \quad y = 5; \quad AB = 2; \quad AC = 6;$

**- Bài tập nâng cao**

3. 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.

**- Tài liệu tham khảo**

1. **Modern Compiler Implementation in C.** A.W. Appel - Cambridge University Press, 1997. Chương 8.
2. **Giáo trình chương trình dịch** 2<sup>nd</sup> Edition. Phạm Hồng Nguyên. NXB ĐHQG Hà Nội, 2009. Chương 7.

**- Câu hỏi ôn tập**

1. Các cách tổ chức bộ nhớ.
2. Các chiến lược cấp phát – thu hồi bộ nhớ