LỜI NÓI ĐẦU

Môn học Chương trình dịch là môn học của ngành khoa học máy tính. Trong suốt thập niên 50, trình biên dịch được xem là cực kỳ khó viết. Ngày nay, việc viết một chương trình dịch trở nên đơn giản hơn. Cùng với sự phát triển của các chuyên ngành lý thuyết ngôn ngữ hình thức và automat, lý thuyết thiết kế một trình biên dịch ngày một hoàn thiện hơn.

Có rất nhiều các trình biên dịch hiện đại, có hỗ trợ nhiều tính năng tiện ích khác. Ví dụ: bộ visual Basic, bộ studio của Microsoft, bộ Jbuilder, netbean, Delphi … Tại sao ta không đứng trên vai những người khổng lồ đó mà lại đi nghiên cứu cách xây dựng một chương trình dịch nguyên thuỷ. Với vai trò là sinh viên công nghệ thông tin ta phải tìm hiểu nghiên cứu xem một chương trình dịch thực sự thực hiện như thế nào?

Mục đích của môn học này là sinh viên sẽ học các thuật toán phân tích ngữ pháp và các kỹ thuật dịch, hiểu được các thuật toán xử lý ngữ nghĩa và tối ưu hóa quá trình dịch.

Yêu cầu người học nắm được các thuật toán trong kỹ thuật dịch.

Nội dung môn học : Môn học Chương trình dịch nghiên cứu 2 vấn đề:

- Lý thuyết thiết kế ngôn ngữ lập trình (cách tạo ra một ngôn ngữ giúp người lập trình có thể đối thoại với máy và có thể tự động dịch được).

- Cách viết chương trình chuyển đổi từ ngôn ngữ lập trình này sang ngôn ngữ lập trình khác.

Học môn Chương trình dịch giúp người học:

- Nắm vững nguyên lý lập trình: Hiểu từng ngôn ngữ, điểm mạnh điểm yếu của nó, từ đó ta có thể chọn ngôn ngữ thích hợp cho dự án của mình. Biết chọn chương trình dịch thích hợp. Phân biệt được công việc nào do chương trình dịch thực hiện và do chương trình ứng dụng thực hiện.

- Vận dụng: thực hiện các dự án xây dựng chương trình dịch. Áp dụng vào các ngành khác như xử lý ngôn ngữ tự nhiên…

Để viết được trình biên dịch ta cần có kiến thức về ngôn ngữ lập trình, cấu trúc máy tính, lý thuyết ngôn ngữ, cấu trúc dữ liệu, phân tích thiết kế giải thuật và công nghệ phần mềm.

Những kiến thức của môn học có thể sử dụng trong các lĩnh vực khác như xử lý ngôn ngữ tự nhiên, dịch Anh Việt…

Chúng tôi biên soạn bài giảng này nhằm mục đích cung cấp một tài liệu dễ đọc dễ hiểu, hỗ trợ cho sinh viên khi học cũng như khi nghiên cứu tìm hiểu về trình biên dịch. Bài giảng được tổng hợp từ các tài liệu và kinh nghiệm giảng dạy của chúng tôi trong những năm qua. Rất mong nhận được sự đóng góp của sinh viên và bạn đọc để bài giảng đ- ợc hoàn thiện hơn.

Thái Nguyên, tháng 2 năm 2009

Các tác giả

CHƯƠNG 1
TỔNG QUAN VỀ CHƯƠNG TRÌNH DỊCH

Mục tiêu: Sinh viên hiểu một cách tổng quan về chương trình dịch và mối quan hệ của nó với các thành phần khác.

Nội dung chính:

- *Mối quan hệ giữa ngôn ngữ lập trình và chương trình dịch.*
- *Khái niệm chương trình dịch, phân loại chương trình dịch.*
- *Cấu trúc của một chương trình dịch.*

1. NGÔN NGỮ LẬP TRÌNH VÀ CHƯƠNG TRÌNH DỊCH

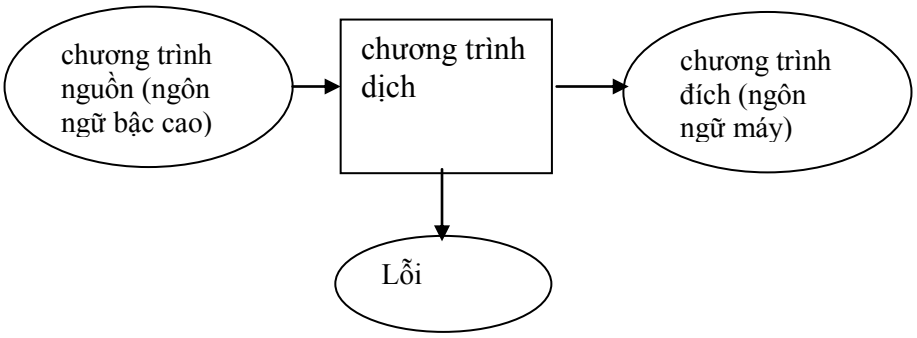
Con người muốn máy tính thực hiện công việc thì con người phải viết yêu cầu đưa cho máy tính bằng ngôn ngữ máy hiểu được. Việc viết yêu cầu gọi là lập trình. Ngôn ngữ dùng để lập trình gọi là ngôn ngữ lập trình. Có nhiều ngôn ngữ lập trình khác nhau. Dựa trên cơ sở của tính không phụ thuộc vào máy tính ngày càng cao người ta phân cấp các ngôn ngữ lập trình như sau:

- Ngôn ngữ máy (machine languge)
- Hợp ngữ (acsembly langguge)
- Ngôn ngữ cấp cao (high level langguage)

Ngôn ngữ máy chỉ gồm các số 0 và 1, khó hiểu đối với người sử dụng. Mà ngôn ngữ tự nhiên của con người lại dài dòng nhiều chi tiết mập mờ, không rõ ràng đối với máy. Để con người giao tiếp được với máy dễ dàng cần một ngôn ngữ trung gian gần với ngôn ngữ tự nhiên. Vì vậy ta cần có một chương trình để dịch các chương trình trên ngôn ngữ này sang mã máy để có thể chạy được. Những chương trình làm nhiệm vụ như vậy gọi là các chương trình dịch. Ngoài ra, một chương trình dịch còn chuyển một chương trình từ ngôn ngữ này sang ngôn ngữ khác tương đương. Thông thường ngôn ngữ nguồn là ngôn ngữ bậc cao và ngôn ngữ đích là ngôn ngữ bậc thấp, ví dụ như ngôn ngữ Pascal hay ngôn ngữ C sang ngôn ngữ Acsembly.

*** Định nghĩa chương trình dịch:**

Chương trình dịch là một chương trình thực hiện việc chuyển đổi một chương trình hay đoạn chương trình từ ngôn ngữ này (gọi là ngôn ngữ nguồn) sang ngôn ngữ khác (gọi là ngôn ngữ đích) tương đương.



Hình 1.1: Sơ đồ một chương trình

2. PHÂN LOẠI CHƯƠNG TRÌNH DỊCH

Có thể phân thành nhiều loại tuỳ theo các tiêu chí khác nhau.

- Theo số lần duyệt: Duyệt đơn, duyệt nhiều lần.

- Theo mục đích: Tải và chạy, gỡ rối, tối ưu, chuyển đổi ngôn ngữ, chuyển đôỉ định dạng…

- Theo độ phức tạp của chương trình nguồn và đích:

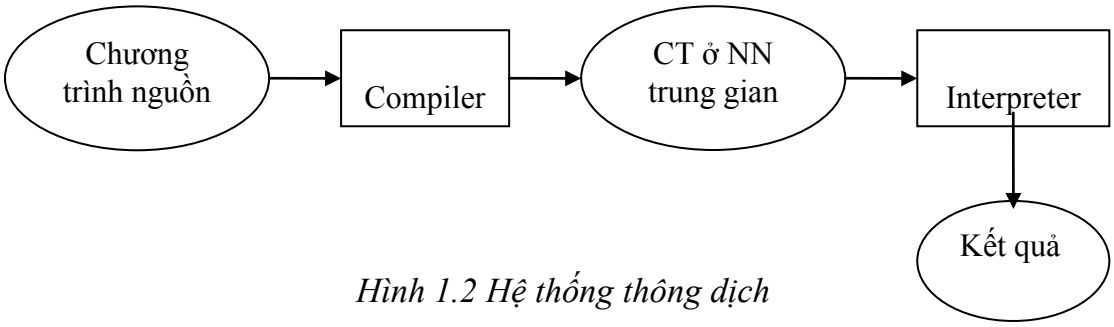
+ Asembler (chương trình hợp dịch): Dịch từ ngôn ngữ asembly ra ngôn ngữ máy.

+ Preproccessor: (tiền xử lý) : Dịch từ ngôn ngữ cấp cao sang ngôn ngữ cấp cao khác (thực chất là dịch một số cấu trúc mới sang cấu trúc cũ).

+ Compiler: (biên dịch) dịch từ ngôn ngữ cấp cao sang ngôn ngữ cấp thấp.

- Theo phương pháp dịch chạy:

+ Thông dịch: (diễn giải - interpreter) chương trình thông dịch đọc chương trình nguồn theo từng lệnh và phân tích rồi thực hiện nó. (Ví dụ hệ điều hành thực hiện các câu lệnh DOS, hay hệ quản trị cơ sở dữ liệu Foxpro). Hoặc ngôn ngữ nguồn không được chuyển sang ngôn ngữ máy mà chuyển sang một ngôn ngữ trung gian. Một chương trình sẽ có nhiệm vụ đọc chương trình ở ngôn ngữ trung gian này và thực hiện từng câu lệnh. Ngôn ngữ trung gian được gọi là ngôn ngữ của một máy ảo, chương trình thông dịch thực hiện ngôn ngữ này gọi là máy ảo.



Hình 1.2 Hệ thống thông dịch

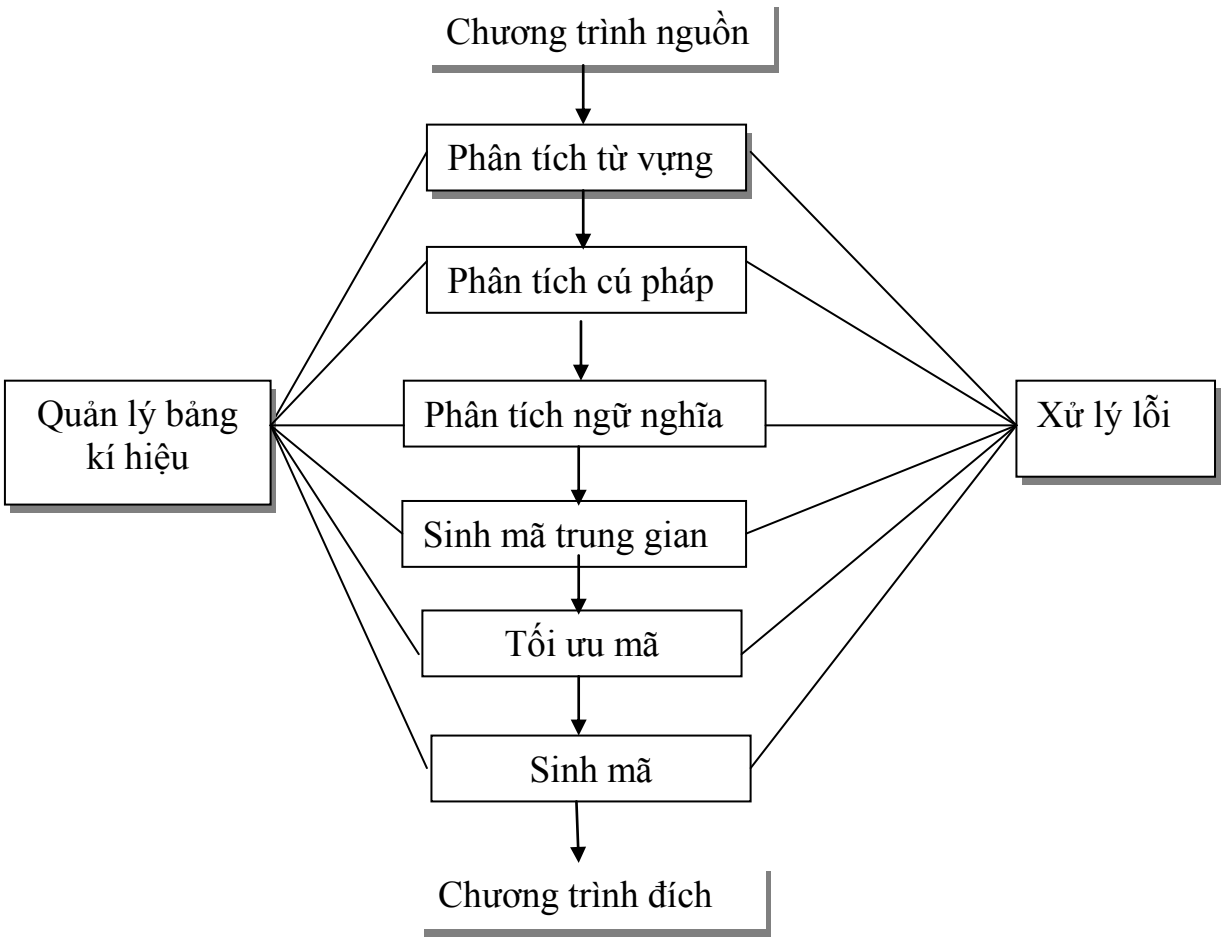
Ví dụ hệ thống dịch Java. Mã nguồn Java được dịch ra dạng Bytecode. File này được một trình thông dịch gọi là máy ảo Java thực hiện.

+ Biên dịch: toàn bộ chương trình nguồn được trình biên dịch chuyển sang chương trình đích ở dạng mã máy. 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 – letf to right, right most)

1.3. Cấu trúc của chương trình dịch

1.3.1. Cấu trúc tĩnh (cấu trúc logic)



Hình 1.3 Cấu trúc tĩnh của chương trình dịch

1) Phân tích từ vựng: đọc luồng kí tự tạo thành chương trình nguồn từ trái sang phải, tách ra thành các từ tố (token).

- Từ vựng: Cũng như ngôn ngữ tự nhiên, ngôn ngữ lập trình cũng được xây dựng dựa trên bộ từ vựng. 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 ví dụ: tên viết bởi chữ cái đầu tiên sau đó là không hoặc nhiều chữ cái hoặc chữ số, phép gán trong C là =, trong Pascal là :=

Để xây dựng một chương trình dịch, hệ thống phải tìm hiểu tập từ vựng của ngôn ngữ nguồn và phân tích để biết được từng loại từ vựng và các thuộc tính của nó.

Ví dụ: Câu lệnh trong chương trình nguồn viết bằng ngôn ngữ pascal:

"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)

2) Phân tích cú pháp: Phân tích cấu trúc ngữ pháp của chương trình. Các từ tố được nhóm lại theo cấu trúc phân cấp.

- Cú pháp: Cú pháp là thành phần quan trọng nhất trong một ngôn ngữ. Như chúng ta đã biết trong ngôn ngữ hình thức thì ngôn ngữ là tập các câu thỏa mãn văn phạm của ngôn ngữ đó. Ví dụ như:

câu = chủ ngữ + vị ngữ

vị ngữ = động từ + bổ ngữ

v.v. . .

Trong ngôn ngữ lập trình, 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. Chúng ta quan tâm đến các cấu trúc này bao gồm:

1) các khai báo
2) biểu thức số học, biểu thức logic
3) các lệnh: lệnh gán, lệnh gọi hàm, lệnh vào ra, . . .
4) câu lệnh điều kiện if
5) câu lệnh lặp: for, while
6) chương trình con (hàm và thủ tục)

Nhiệm vụ trước tiên là phải biết được bộ luật cú pháp của ngôn ngữ mà mình định xây dựng chương trình cho nó.

Với một chuỗi từ tố và tập luật cú pháp của ngôn ngữ, bộ phân tích cú pháp 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 bộ phân tích cú pháp áp dụng tất cả các luật nhưng không thể xây dựng được cây cú pháp của chuỗi nhập thì thông báo rằng chuỗi nhập không viết đúng cú pháp.

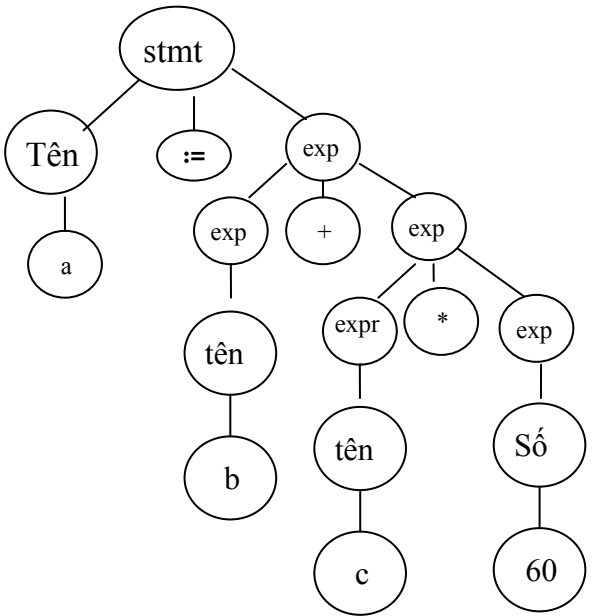
Chương trình phải phân tích chương trình nguồn thành các cấu trúc cú pháp của ngôn ngữ, từ đó để kiểm tra tính đúng đắn về mặt ngữ pháp của chương trình nguồn.

Ví dụ: Ngôn ngữ được đặc tả bởi luật sau:

Stmt → ten := expr

Expr → expr + expr | expr * expr | ten | so

Với xâu nhập " a:= b+c*60" ta có cây suy dẫn như sau:



3) Phân tích ngữ nghĩa: 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.

Dựa trên cây cú pháp bộ phân tích ngữ nghĩa xử lý từng phép toán. Mỗi

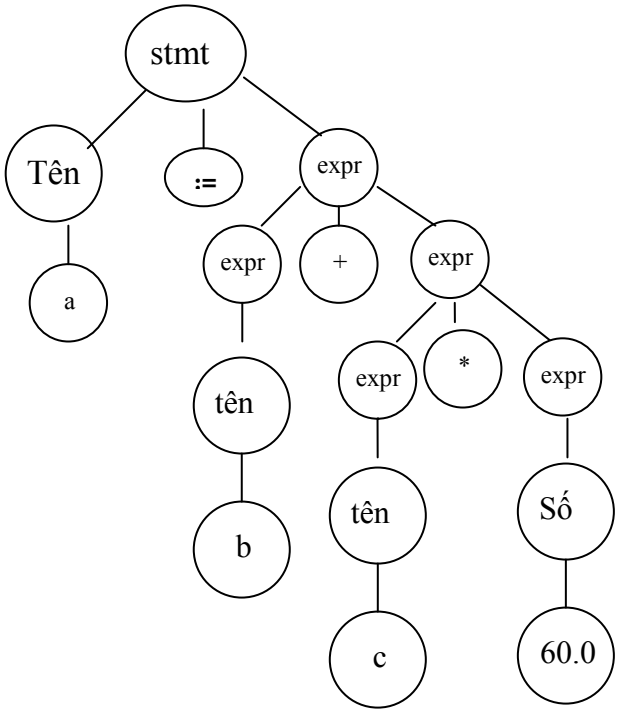
phép toán nó kiểm tra các toán hạng và loại dữ liệu của chúng có phù hợp với phép toán không.

VD: tên (biến) được khai báo kiểu real, 60 là kiểu interge vì vậy trình biên dịch đổi thành số thực 60.0.

- Ngữ nghĩa: của một ngôn ngữ lập trình liên quan đến:

+ Kiểu, phạm vi của hằng và biến

+ Phân biệt và sử dụng đúng tên hằng, tên biến, tên hàm



Chương trình dịch phải kiểm tra tính đúng đắn trong sử dụng các đại lượng này. Ví dụ kiểm tra 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) . . .

4) Sinh mã trung gian: Sinh chương trình trong ngôn ngữ trung gian nhằm: dễ sinh và tối ưu mã hơn dễ chuyển đổi về mã máy hơn.

sau giai đoạn phân tích thì mã trung gian sinh ra như sau:

```
temp1 := 60
temp2 := id3 * temp1
temp3 := id2 + temp 2
id1 := temp3                (1.2)
```

5) Tối ưu mã: Sửa đổi chương trình trong ngôn ngữ trung gian nhằm cải tiến chương trình đích về hiệu năng.

Ví dụ như với mã trung gian ở (1.2), chúng ta có thể làm tốt hơn đoạn mã để tạo ra được các mã máy chạy nhanh hơn như sau:

```
temp1 := id3 * 60
id1 := id2 + temp1 (1.3)
```

6) Sinh mã: tạo ra chương trình đích từ chương trình trong ngôn ngữ trung gian đã tối ưu.

Thông thường là sinh ra mã máy hay mã hợp ngữ. Vấn đề quyết định là việc gán các biến cho các thanh ghi.

Chẳng hạn sử dụng các thanh ghi R1 và R2, các chỉ thị lệnh MOVF, MULF, ADDF, chúng ta sinh mã cho (1.3) như sau:

```
MOVF id3, R2
MULF #60, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1          (1.4)
```

Ngoài ra, chương trình dịch còn phải thực hiện nhiệm vụ:

* ***Quản lý bảng ký hiệu:*** 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 ... để dùng cho các bước cần đến.

Từ tố (token) + thuộc tính (kiểu, phạm vị …)= 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.

- Giai đoạn phân tích từ vựng: lưu trữ trị từ vựng vào bảng kí hiệu nếu nó chưa có.

- 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 phần tử là một mẩu tin dùng để lưu trữ trị từ vựng và các thuộc tính của nó.

- Trị từ vựng: tên từ tố.

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

**** 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 dầu 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.

* Giai đoạn phân tích có đầu vào là ngôn ngữ nguồn, đầu ra là ngôn ngữ trung gian gọi là kỳ trước (fron end). Giai đoạn tổng hợp có đầu vào là ngôn ngữ trung gian và đầu ra là ngô ngữ đích gọi là kỳ sau (back end).

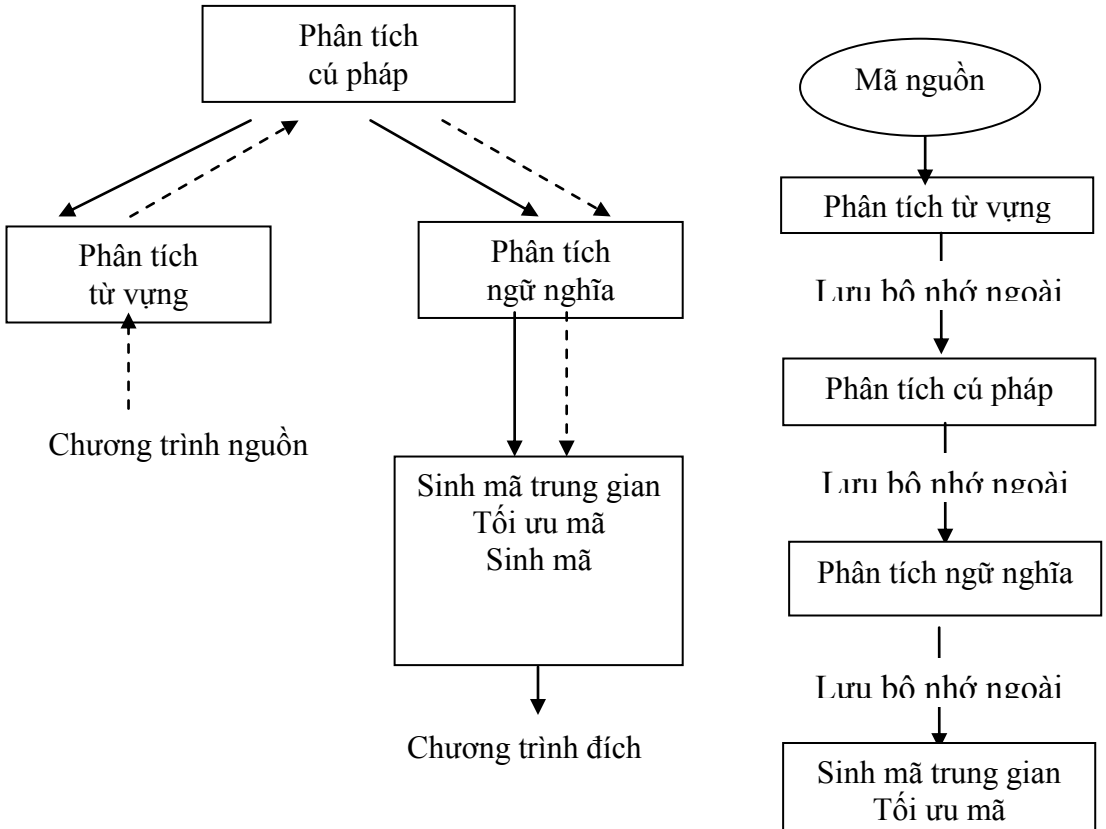
Đối với các ngôn ngữ nguồn, ta chỉ cần quan tâm đến việc sinh ra mã trung gian mà không cần biết mã máy đích của nó. Điều này làm cho công việc đơn giản, không phụ thuộc vào máy đích. Còn giai đoạn sau trở nên đơn giản hơn vì ngôn ngữ trung gian thường thì gần với mã máy. Và nó còn thể hiện ưu điểm khi chúng ta xây dựng nhiều cặp ngôn ngữ.

3.2. Cấu trúc động.

Cấu trúc động (cấu trúc theo thời gian) cho biết quan hệ giữa các phần khi hoạt động.

Các thành phần độc lập của chương trình có thể hoạt động theo 2 cách: lần lượt hay đồng thời. mỗi khi một phần nào đó của chương trình dịch xong toàn bộ chương trình nguồn hoặc chương trình trung gian thì ta gọi đó là một lần duyệt.

* Duyệt đơn (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. Nó gọi bộ phân tích từ vựng khi cần một từ tố tiếp theo và gọi bộ phân tích ngữ nghĩa khi muốn chuyển cho một cấu trúc cú pháp đã được phân tích. Bộ phân tích ngữ nghĩa lại đưa cấu trúc sang phần sinh mã trung gian để sinh ra các mã trong một ngôn ngữ trung gian rồi đưa vào bộ tối ưu và sinh mã.



HÌnh 1.4 Chương trình dịch duyệt đơn

Hình 1.5 Chương trình dịch duyệt nhiều lần

* 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 ngaòi để lại được đọc vào cho bước tiếp theo.

Người ta chỉ muốn có một số ít lượt bởi vì mỗi lượt đều mất thời gian đọc và ghi ra tập tin trung gian. Ngược lại nếu gom quá nhiều giai đoạn vào trong một lượt thì phải duy trì toàn bộ chương trình trong bộ nhớ, vì 1 giai đoạn cần thông tin theo thứ tự khác với thứ tự nó được tạo ra. Dạng biểu diễn trung gian của chương trình lớn hơn nhiều so với ct nguồn hoặc ct đích, nên sẽ gặp vấn đề về bộ nhớ.

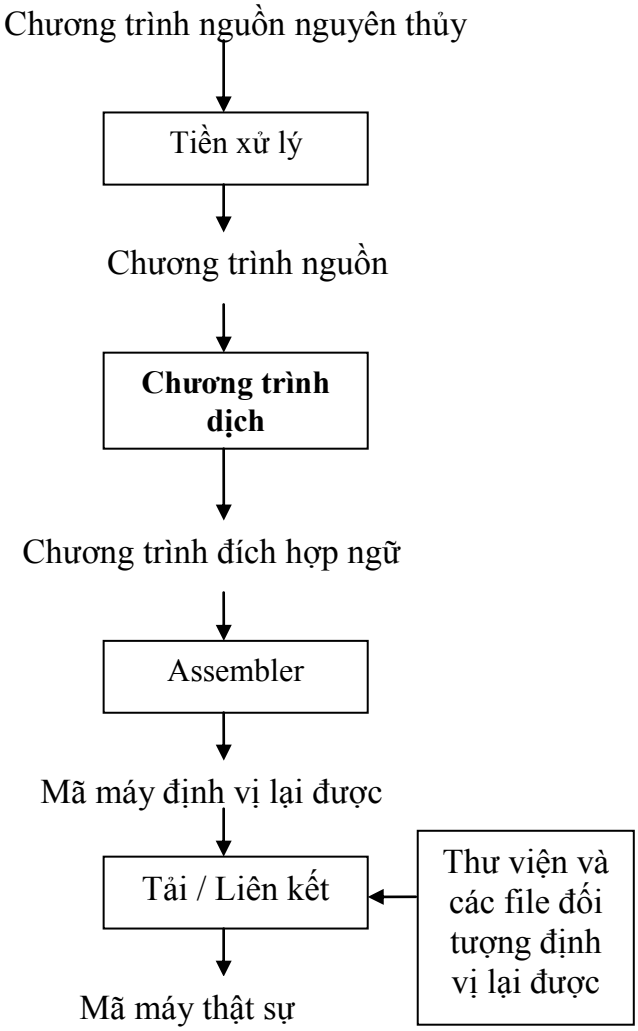
Ưu và nhược điểm của các loại:

So sánh	duyệt đơn	duyệt nhiều lần
tốc độ	tốt	Kém
bộ nhớ	kém	tốt
độ phức tạp	kém	tốt
Các ứng dụng lớn	Kém	tốt

Trong giáo trình này chúng ta nghiên cứu các giai đoạn của một chương trình dịch một cách riêng rẽ nhưng theo thiết kế duyệt một lượt.

4. MÔI TRƯỜNG BIÊN DỊCH

Chương trình dịch là 1 chương trình trong hệ thống liên hoàn giúp cho người lập trình có được một môi trường hoàn chỉnh để phát triển các ứng dụng của họ. Chương trình dịch trong hệ thống đó thể hiện trong sơ đồ sau:



Hình 1.6 Hệ thống xử lý ngôn ngữ

* Bộ tiền xử lý:

Chuỗi kí tự nhập vào chương trình dịch là các kí tự của chương trình nguồn nhưng trong thực tế, trước khi là đầu vào của một chương trình dịch, toàn bộ file nguồn sẽ được qua một thậm chí một vài bộ tiền xử lý. Sản phẩm của các bộ tiền xử lý này mới là chương trình nguồn thực sự của chương trình dịch. Bộ tiền xử lý sẽ thực hiện các công việc sau:

- Xử lý Macro: Cho phép người dùng định nghĩa các macro là cách viết tắt của các cấu trúc dài hơn.

- Chèn tệp tin: Bổ sung nội dung của các tệp tin cần dùng trong chương trình. Ví dụ: Trong ngôn ngữ Pascal có khai báo thư viện "Uses crt;"

Bộ tiền xử lý sẽ chèn tệp tin crt vào thay cho lời khai báo.

- Bộ xử lý hoà hợp: hỗ trợ những ngôn ngữ xưa hơn bằng các cấu trúc dữ liệu hoặc dòng điều khiển hiện đại hơn.

- Mở rộng ngôn ngữ: gia tăng khả năng của ngôn ngữ bằng các macro có sẵn.

* Trình biên dịch hợp ngữ: Dịch các mã lệnh hợp ngữ thành mã máy.

* Trình tải/ liên kết:

Trình tải nhận các mã máy khả tải định vị, thay đổi các địa chỉ khả tải định vị, đặt các chỉ thị và dữ liệu trong bộ nhớ đã được sửa đổi vào các vị trí phù hợp.

Trình liên kết cho phép tạo ra một chương trình từ các tệp tin thư viện hoặc nhiều tệp tin mã máy khả tải định vị mà chúng là kết quả của những biên dịch khác nhau.

CHƯƠNG 2
PHÂN TÍCH TỪ VỰNG

Mục tiêu:

Sinh viên hiểu và nhận biết được được các từ tố, biểu diễn được các từ tố trong máy tính và xây dựng chương trình đoán nhận từ tố đó.

Nội dung: *Kỹ thuật xác định và xây dựng bộ phân tích từ vựng*

1. QUÁ TRÌNH PHÂN TÍCH TỪ VỰNG

1) Xóa bỏ kí tự không có nghĩa *(các chú thích, dòng trống, kí hiệu xuống dòng, kí tự trống không cần thiết)*

Quá trình dịch sẽ xem xét tất cả các ký tự trong dòng nhập nên những ký tự không có nghĩa (khoảng trắng (blanks, tabs, newlines) hoặc lời chú thích phải bị bỏ qua. Khi bộ phân tích từ vựng bỏ qua các khoảng trắng này thì bộ phân tích cú pháp không bao giờ quan tâm đến nó nữa.

2) Nhận dạng các kí hiệu: nhận dạng các từ tố.

Ví dụ ghép các chữ số để được một số và sử dụng nó như một đơn vị trong suốt quá trình dịch. Đặt num là một token biểu diễn cho một số nguyên. Khi một chuỗi các chữ số xuất hiện trong dòng nhập thì bộ phân tích sẽ gửi cho bộ phân tích cú pháp num. Giá trị của số nguyên đã được chuyển cho bộ phân tích cú pháp như là một thuộc tính của token num.

3) Số hoá các kí hiệu: Do con số xử lý dễ dàng hơn các xâu, từ khoá, tên, nên xâu thay bằng số, các chữ số được đổi thành số thực sự biểu diễn trong máy. Các tên được cất trong danh sách tên, các xâu cất trong danh sách xâu, các chuỗi số trong danh sách hằng số.

2. TỪ VỊ, TỪ TỐ, MẪU.

* Từ vị (lexeme): là một nhóm các kí tự kề nhau có thể tuân theo một quy ước (mẫu hay luật) nào đó.

* Từ tố (token): là một thuật ngữ chỉ các từ vựng có cùng ý nghĩa cú pháp (cùng một luật mô tả).

- Đối với ngôn ngữ lập trình thì từ tố có thể được phân vào các loại sau: từ khoá, tên (tên của hằng, hàm, biến), số, xâu ký tự, các toán tử, các ký hiệu.

Ví dụ: position := initial + 10 * rate ;

ta có các từ vựng position, :=, initial, +, 10, *, rate, ;

Trong đó position, initial, rate là các từ vựng có cùng ý nghĩa cú pháp là các tên.

:= là phép gán

+ là phép cộng

* là phép nhân

10 là một con số

; là dấu chấm phẩy

Như vậy trong câu lệnh trên có 8 từ vựng thuộc 6 từ tố.

Phân tích cú pháp sẽ làm việc trên các từ tố chứ không phải từ vựng, ví dụ như là làm việc trên khái niệm một số chứ không phải trên 5 hay 2; làm việc trên khái niệm tên chứ không phải là a, b hay c.

* Thuộc tính của từ tố:

Một từ tố có thể ứng với một tập các từ vị khác nhau, ta buộc phải thêm một số thông tin nữa để khi cần có thể biết cụ thể đó là từ vị nào. Ví dụ: 15 và 267 đều là một chuỗi số có từ tố là num nhưng đến bộ sinh mã phải biết cụ thể đó là số 15 và số 267.

Thuộc tính của từ tố là những thông tin kết hợp với từ tố đó. Trong thực tế, một từ tố sẽ chứa một con trỏ trỏ đến một vị trí trên bảng kí hiệu có chứa thông tin về nó.

Ví dụ: position := initial + 10 * rate ; ta nhận được dãy từ tố:

<tên, con trỏ trỏ đến position trên bảng kí hiệu>

<phép gán, >

<tên, con trỏ trỏ đến initial trên bảng kí hiệu>

<phép cộng, >

<tên, con trỏ trỏ đến rate trên bảng kí hiệu>

<phép nhân>

<số nguyên, giá trị số nguyên 60>

* Mẫu (luật mô tả - patter): Để cho bộ phân tích từ vựng nhận dạng được các từ tố, thì đối với mỗi từ tố chúng ta phải mô tả đặc điểm để xác định một từ vựng có thuộc từ tố đó không, mô tả đó được gọi là mẫu từ tố hay luật mô tả.

Token	**Trị từ vựng**	**Mẫu (luật mô tả)**
const	const	const
if	if	if

quan hệ (relation)	<,<=,=,<>,>,>=	< hoặc <= hoặc =hoặc <> hoặc <> hoặc > hoặc >=
tên	pi, count, d2	mở đầu là chữ cái theo sau là chữ cái, chữ số
Số (num)	3.1416, 0, 5	bất kỳ hằng số nào
Xâu (literal)	'hello'	bất kỳ các character nằm giữa ' và ' ngoại trừ '

Trị từ vựng được so cùng với mẫu của từ tố là chuỗi kí tự và là đơn vị của từ vựng. Khi đọc chuỗi kí tự của chương trình nguồn bộ phân tích từ vựng sẽ so sánh chuỗi kí tự đó với mẫu của từ tố nếu phù hợp nó sẽ đoán nhận được từ tố đó và đưa từ tố vào bảng kí hiệu cùng với trị từ vựng của nó.

3. CÁCH LƯU TRỮ CHƯƠNG TRÌNH NGUỒN

Việc đọc từng kí tự trong chương trình nguồn tốn một thời gian đáng kể nên nó ảnh hưởng tới tốc độ chương trình dịch. Để giải quyết vấn đề này, thiết kế đọc vào một lúc một chuỗi kí tự lưu trữ vào vùng nhớ tạm buffer. Nhưng việc đọc như vậy gặp khó khăn do không thể xác định được một chuỗi như thế nào thì chứa chọn vẹn 1 từ tố. Và phải phân biệt được một chuỗi như thế nào thì chứa chọn vẹn một từ tố.Có 2 phương pháp giải quyết như sau:

3.1 Cặp bộ đệm (buffer pairs)

* Cấu tạo:

- Chia buffer thành 2 nửa, một 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: (forwar):di chuyển trên từng kí tự trong buffer để xác định từ tố.

	E	=	M	*	C	*	*	2	EOF		

** Hoạt động:*

- Đọc n kí tự vào nửa đầu của buffer, 2 con trỏ trùng nhau tại vị trí bắt đầu.

- Con trỏ p2 tiến sang phải cho tới khi xác định được một từ tố có từ vị là chuỗi kí tự nằm giữa 2 con trỏ. Dời p1 lên trùng với p2, tiếp tục dò tìm từ tố mới.

- Khi p2 ở cuối nửa đầu của buffer thì đọc tiếp n kí tự vào nửa đầu thứ 2. Khi p2 nằm ở nửa cuối của buffer thì đọc tiếp n kí tự vào nửa đầu của buffer và p2 được dời về đầu của bộ đệm.

- Nếu số kí tự trong chương trình nguồn còn lại ít hơn n thì một kí tự đặc biệt được đưa vào buffer sau các kí tự vừa đọc để báo hiệu chương trình nguồn đã được đọc hết.

3.2 Phương pháp cầm canh.

Phương pháp trên mỗi lần di chuyển p2 phải kiểm tra xem có phải đã hết một nửa buffer chưa nên kém hiệu quả vì phải 2 lần test. Khắc phục:

- Mỗi lần chí đọc n-1 kí tự vào mỗi nửa buffer còn kí tự thứ n là kí tự đặc biệt (thường là EOF). Như vậy ta chỉ cần một lần test.

	E	=	M	*	EOF		C	*	*	2	EOF		EOF

Giải thuật:

```
p2 := p2 + 1;
if    p2 = eof    then
        begin
          if    p2 ở cuối của nửa đầu    then
                begin
                        Đọc vào nửa cuối;  p2 := p2 + 1
                end
          else   if   p2 ở cuối của nửa cuối   then
                begin
                Đọc vào nửa đầu;  Dời p2 vào đầu của nửa đầu
                End
             else  /* eof ở giữa chỉ hết chơng trình nguồn */
                kết thúc phân tích từ vựng
        end;
```

4. BIỂU DIỄN TỪ TỐ

Cách biểu diễn các luật đơn giản nhất là biểu diễn bằng lời. Tuy nhiên cách này thường gặp hiện tượng nhập nhằng (cùng một lời nói có thể hiểu theo nhiều nghĩa khác nhau), phát biểu theo nhiều cách khác nhau khó đưa vào máy tính. Các từ tố khác nhau có các mẫu hay luật mô tả khác nhau. Các mẫu này là cơ sở để nhận dạng các từ tố. Ta cần thiết phải hình thức hoá các mẫu này để làm sao có thể lập trình được. Việc này có thể thực hiện được nhờ biểu thức chính qui và ôtômát hữu hạn. Ngoài ra ta có thể dùng cách biểu diễn trực quan của văn phạm phi ngữ cảnh là đồ thị chuyển để mô tả các loại từ tố.

4.1. Một số khái niệm về ngôn ngữ hình thức

* *Bảng chữ cái*: là một tập $\Sigma \neq \varnothing$ hữu hạn hoặc vô hạn các đối tượng. Mỗi phần tử a $\in\Sigma$ gọi là kí hiệu hoặc chữ cái (thuộc bảng chữ cái Σ).

* *Xâu:* Là một dãy liên tiếp các kí hiệu thuộc cùng một bảng chữ cái.

- Độ dài xâu: là tổng vị trí của tất cả các kí hiệu có mặt trong xâu, kí hiệu là |w|.

- Xâu rỗng: là từ có độ dài = 0 kí hiệu là ε hoặc $\wedge$. Độ dài = 0.

- Xâu v là **Xâu con** của w nếu v được tạo bởi các ký hiệu liền kề nhau trong w.

* Tập tất cả các từ trên bảng chữ cái Σ kí hiệu là Σ^*. Tập tất cả các từ khác rỗng trên bảng chữ cái Σ kí hiệu là Σ^+. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

* *Tiền tố*: của một xâu là một xâu con bất kỳ nằm ở đầu xâu. H**ậu tố** của một xâu là xâu con nằm ở cuối xâu. (Tiền tố và hậu tố của một xâu khác hơn chính xâu đó ta gọi là tiền tố và hậu tố thực sự)

* *Ngôn ngữ*: Một ngôn ngữ L là một tập các chuỗi của các ký hiệu từ một bộ chữ cái Σ nào đó. (Một tập con $A \subseteq \Sigma^*$ được gọi là một ngôn ngữ trên bảng chữ cái Σ).

- Tập rỗng được gọi là ngôn ngữ trống (hay ngôn ngữ rỗng). Ngôn ngữ rỗng là ngôn ngữ trên bất kỳ bảng chữ cái nào. (Ngôn ngữ rỗng khác ngôn ngữ chỉ gồm từ rỗng: ngôn ngữ $\varnothing$ không có phần tử nào trong khi ngôn ngữ $\{\varepsilon\}$ có một phần tử là chuỗi rỗng ε)

* *Các phép toán trên ngôn ngữ.*

+ Phép giao: $L = L_1 \cap L_2 = \{x \in\Sigma^* \mid x\in L_1 \text{ hoặc } x \in L_2\}$

+ Phép hợp: $L = L_1 \cup L_2 = \{x \in\Sigma^* \mid x\in L_1 \text{ và } x \in L_2\}$

+ Phép lấy phần bù của ngôn ngữ L là tập $CL = \{ x \in\Sigma^* \mid x \notin L\}$

$$\overline{L} = \Sigma^* - L$$

+ Phép nối kết (concatenation) của hai ngôn ngữ L_1/ Σ_1 và L_2/Σ_2 là :

$$L_1L_2 = \{w_1w_2 \mid w_1\in L_1 \text{ và } w_2 \in L_2 \}/ \Sigma_1 \cup \Sigma_2$$

Ký hiệu $L^n = L.L.L\ldots L$ (n lần). $L^i = LL^{i-1}$.

- Trường hợp đặc biệt : $L^0 = \{\varepsilon\}$, với mọi ngôn ngữ L.

+ Phép bao đóng (closure) :

+ Bao đóng *(Kleene)* của ngôn ngữ L, ký hiệu L^* là hợp của mọi tập tích trên L:

$$L^* = \infty \cup_{Ii=0} Li$$

+ Bao đóng dương *(positive)* của ngôn ngữ L, ký hiệu L^+ được định nghĩa là hợp của mọi tích dương trên L : L: $L^+ = \infty \cup_{i=1} L^I$

* *Văn phạm*

Định nghĩa văn phạm. (văn phạm sinh hay văn phạm ngữ cấu)

- Là một hệ thống gồm bốn thành phần xác định **G = (Σ, Δ, P, S)**, trong đó:

Σ : tập hợp các ký hiệu kết thúc (**t**erminal).

Δ : tập hợp các biến hay ký hiệu chưa kết thúc (với $\Sigma \cap \Delta = \varnothing$)

P : tập hữu hạn các quy tắc ngữ pháp được gọi là các sản xuất (**p**roduction), mỗi sản xuất biểu diễn dưới dạng $\alpha \rightarrow \beta$, với α, β là các chuỗi $\in (\Sigma \cup \Delta)^*$.

S $\subset \Delta$: ký hiệu chưa kết thúc dùng làm ký hiệu bắt đầu (**s**tart)

Quy ước:

- Dùng các chữ cái Latinh viết hoa (A, B, C, ...) để chỉ các ký hiệu trong tập biến Δ.

- Các chữ cái Latinh đầu bảng viết thường (a, b, c, ...) chỉ ký hiệu kết thúc thuộc tập Σ

- Xâu thường được biểu diễn bằng các chữ cái Latinh cuối bảng viết thường (x, y, z, ...).

* *Phân loạivăn phạm theo Chosmky*.

- Lớp 0: là *văn phạm ngữ cấu* (Phrase Structure) với các luật sản xuất có dạng: α -> β với $\alpha \in V^+, \beta \in V^*$

- Lớp 1: là *văn phạm cảm ngữ cảnh* (Context Sensitive) với các luật sản xuất có dạng: α -> β với $\alpha \in V^+, \beta \in V^*$, $|\alpha| < |\beta|$

- Lớp 2: là *văn phạm phi ngữ cảnh* (Context Free Grammar - CFG) với các luật sản xuất có dạng: A -> α với $A \in N, \alpha \in V^*$

- Lớp 3: là *văn phạm chính qui* (Regular Grammar) với luật sản xuất có dạng: A -> a, A -> Ba hoặc A-> a, A-> aB với $A, B \in N$ và $a \in T$

Các lớp văn phạm được phân loại theo thứ tự phạm vi biểu diễn ngôn ngữ giảm dần, lớp văn phạm sau nằm trong phạm vi của lớp văn phạm trước:

Lớp 0 ∈ Lớp 1 ∈ Lớp 2 ∈ Lớp 3

* *Văn phạm chính quy và biểu thức chính quy*.

Ví dụ 1: Tên trong ngôn ngữ Pascal là một từ đứng đầu là chữ cái, sau đó có thể là không hoặc nhiều chữ cái hoặc chữ số.

Biểu diễn bằng BTCQ: tên -> chữ_cái (chữ_cái | chữ_số)*

Biểu diễn bằng văn phạm chính qui:

Tên -> chữ_cái A; A -> chữ_cái A | chữ_số A | ε

* *Biểu thức chính qui* được định nghĩa trên bộ chữ cái ∑ như sau:

- ε là biểu thức chính quy, biểu thị cho tập {ε}

- a ∈ ∑, a là biểu thức chính quy, biểu thị cho tập {a}

- Giả sử r là biểu thức chính quy biểu thị cho ngôn ngữ L(r), s là biểu thức chính quy, biểu thị cho ngôn ngữ L(s) thì:

+ (r)|(s) là biểu thứcchính quy biểu thị cho tập ngôn ngữ L(r) ∪ L(s)

+ (r)(s) là biểu thức chính quy biểu thị cho tập ngôn ngữ L(r)L((s)

+ (r)* là biểu thức chính quy biểu thị cho tập ngôn ngữ L(r)*

Biểu thức chính quy sử dụng các ký hiệu sau:

	là ký hiệu hoặc (hợp)
( )	là ký hiệu dùng để nhóm các ký hiệu
*	là lặp lại không hoặc nhiều lần
+	là lặp lại một hoặc nhiều lần
!	là lặp lại không hoặc một lần

* *Ôtômát hữu hạn:* Một Otomat hữu hạn đơn định là một hệ thống M = (∑, Q, δ, q_0, F), trong đó:

- ∑ là một bộ chữ hữu hạn, gọi là bộ chữ vào
- Q là một tập hữu hạn các trạng thái
- q_0 ∈ Q là trạng thái đầu
- F ∈ Q là tập các trạng thái cuối

δ là hàm chuyển trạng thái δ có dạng:

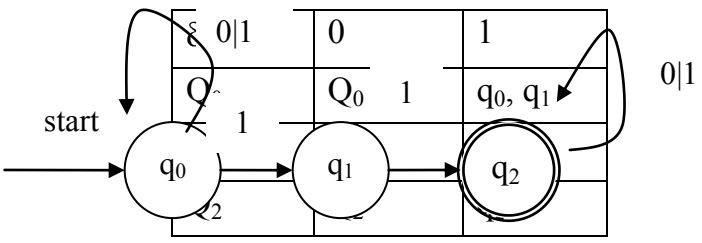
- δ: Q x ∑ -> Q thì M gọi là ôtômát mát đơn định (kí hiệu ÔHĐ).
- δ: Q x ∑ -> 2^Q thì M gọi là ôtômát không đơn định (kí hiệu ÔHK).

* Hình trạng: của một OHĐ là một xâu có dạng qx với q ∈ Q là trạng thái hiện thời và x ∈ $\sum^*$ là phần xâu vào chưa được đoán nhận.

Ví dụ:

∑ = {0, 1}; Q = {q_0, q_1, q_2}; q_0 là trạng thái ban đầu; F={q_2}.

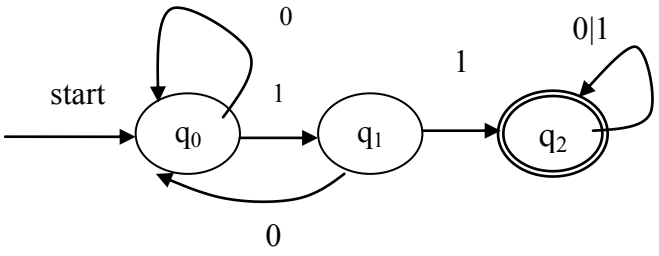
Hàm chuyển trạng thái được mô tả như bảng sau:(ÔHK)



Đồ thị chuyển không đơn định

Hàm chuyển trạng thái ÔHĐ

δ	0	1
Q_0	Q_0	q_1
Q_1	Q_0	q_2
Q_2	Q_2	q_2

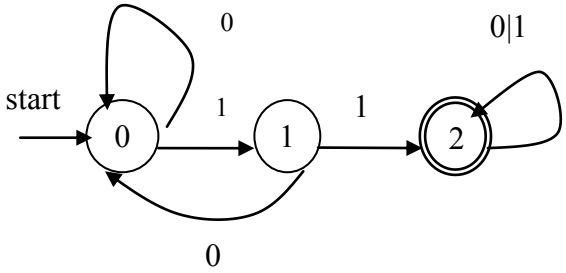


Đồ thị chuyển đơn định

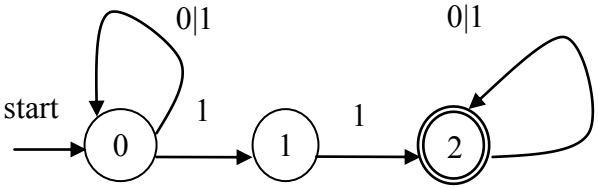
Ví dụ: Viết biểu thức chính qui và đồ thị chuyển để biểu diễn các xâu gồm các chữ số 0 và 1, trong đó tồn tại ít nhất một xâu con "11"

Biểu thức chính qui: (0|1)*11(0|1)*

Biểu diễn biểu thức chính quy dưới dạng đồ thị chuyển:



Đồ thị chuyển đơn định



Đồ thị chuyển không đơn định

4.2. Một số từ tố được mô tả bằng lời như sau:

- Tên là một xâu bắt đầu bởi một chữ cái và theo sau là không hoặc nhiều chữ cái hoặc chữ số
- Số nguyên bao gồm các chữ số
- Số thực có hai phần: phần nguyên và phần thực là xâu các chữ số và hai phần này cách nhau bởi dấu chấm
- Các toán tử quan hệ <, <=, >, >=, <>, =

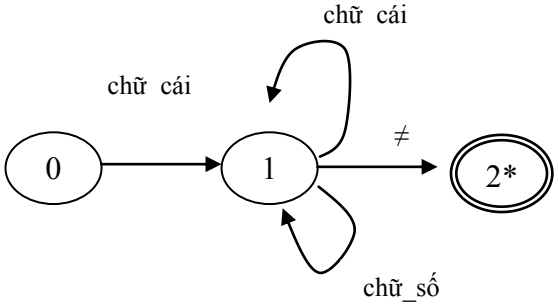
4.3. Mô tả các mẫu từ tố trên bằng biểu thức chính qui:

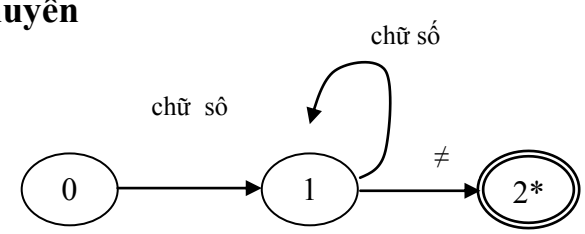
Tên từ tố → biểu thức chính quy biểu diễn từ tố đó.

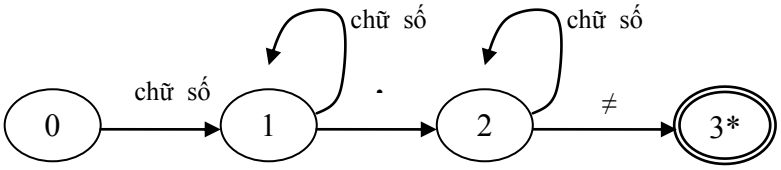
- chữ_cái → A|B|C|…|Z|a|b|c|…|z
- chữ_số → 0|1||2|3|4|5|6|7|8|9
- Tên → *chữ_cái (chữ_cái | chữ_số)**
- Số nguyên → *(chữ_số)*$^+$
- Số thực → *(chữ_số)*$^+$*.(chữ_số)*
- Toán tử quan hệ:

+ Toán tử bé hơn (LT): <
+ Toán tử bé hơn hoặc bằng (LE): <=
+ Toán tử lớn hơn (GT): >
+ Toán tử lớn hơn hoặc bằng (GE): >=
+ Toán tử bằng (EQ): =
+ Toán tử khác (NE): <>

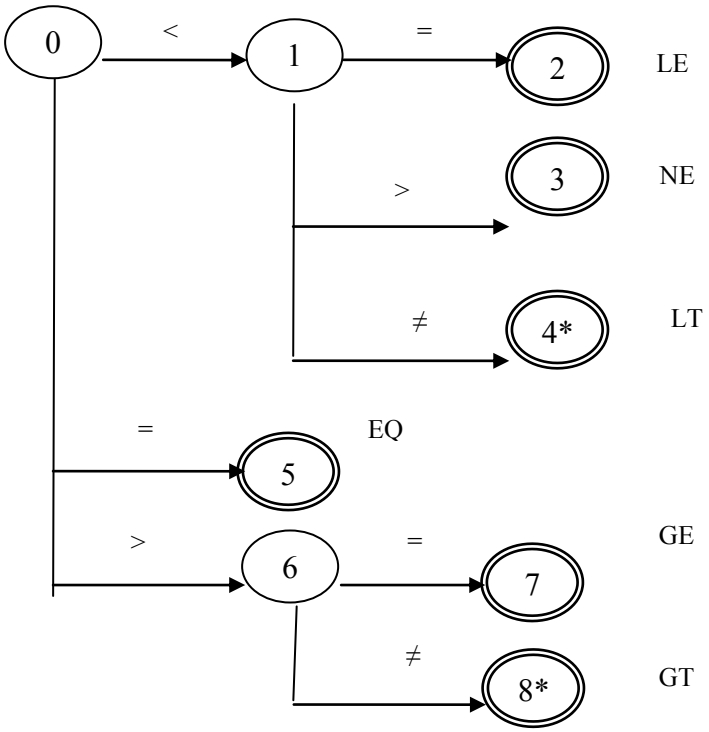
4.4. Biểu diễn từ tố bằng đồ thị chuyển



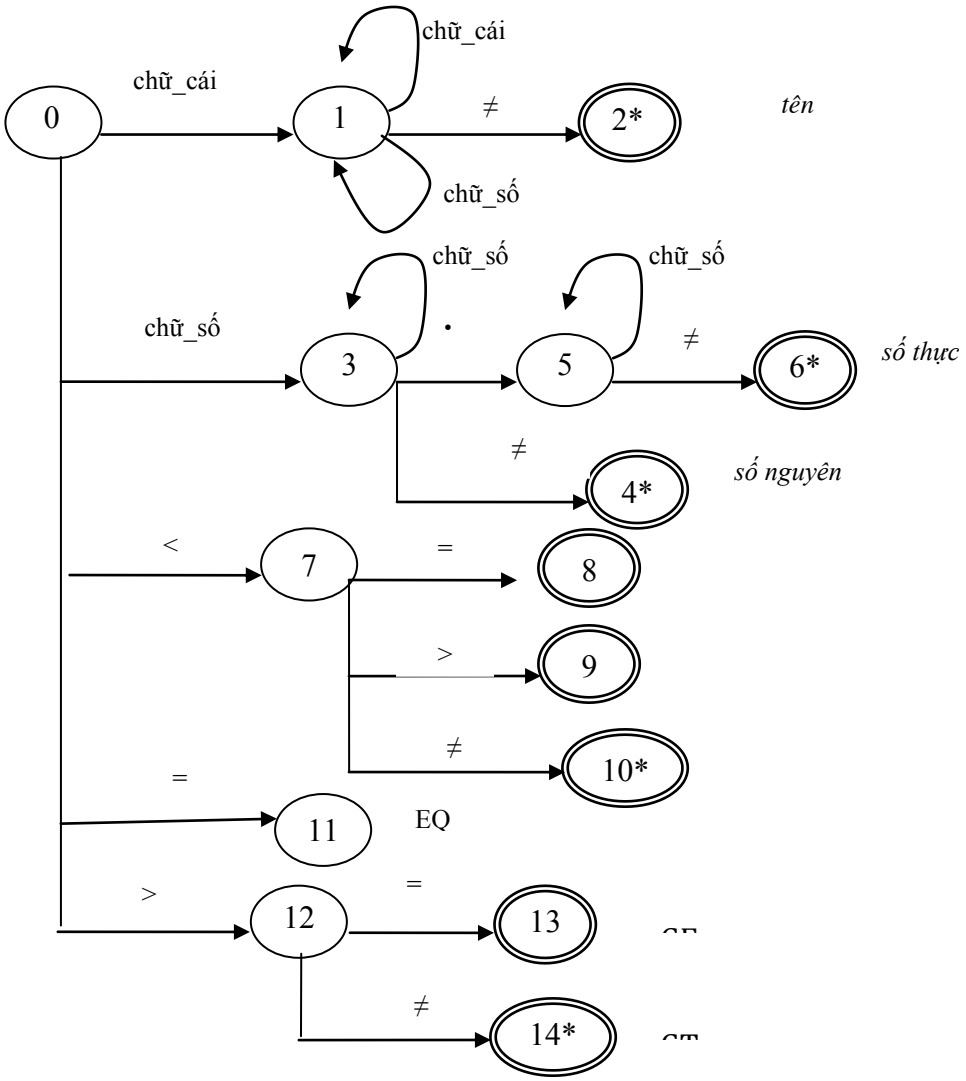




Toán tử quan hệ:



Để xây dựng một chương trình nhận dạng tất cả các loại từ tố này, chúng ta phải kết hợp các đồ thị này thành một đồ thị duy nhất:



4.5. Biểu diễn bởi bảng chuyển

Với ví dụ trên chúng ta xây dựng ôtômát với các thông số như sau:

Q = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14}

F = {2,4,6,10,14}

$q_0 = 0$

hàm chuyển trạng thái được mô tả bởi bảng sau:

∂	**chữ_cái**	**chữ_số**	**.**	**<**	**=**	**>**	**khác**
0	1	3	lỗi	7	11	12	lỗi
1	1	1	2	2	2	2	2
3	4	3	5	4	4	4	4
5	6	5	6	6	6	6	6
7	10	10	10	10	8	9	10
12	14	14	14	14	13	14	14

Các trạng thái ∈ F là trạng thái kết thúc

Các trạng thái có dấu * là kết thúc trả về ký hiệu cuối cho từ tố tiếp theo

5. VIẾT CHƯƠNG TRÌNH CHO BỘ PHÂN TÍCH TỪ VỰNG

5.1. Lập bộ phân tích từ vựng theo đồ thị chuyển.

Đoạn chương trình mô tả việc nhận dạng từ tố bằng cách diễn giải đồ thị chuyển.

Chúng sẽ sử dụng các hàm sau:.

```
int IsDigit ( int c); // hàm kiểm tra một ký hiệu là chữ số
int IsLetter ( int c); // hàm kiểm tra một ký hiệu là chữ cái
int GetNextChar(); // hàm lấy ký tự tiếp theo
```

```
enum Token {IDENT, INTEGER, REAL, LT, LE, GT, GE, NE, EQ, ERROR};
// hàm này trả về loại từ tố
// từ vị nằm trong s
Token  GetNextToken(char *s)
{int state=0;
  int i=0;
  while(1)
  {       int c=GetNextChar();
          switch(state)
          {       case 0:         if(IsLetter(c)) state=1;
```

```
                    else if(IsDigit(c)) state=3;
                                else if(c==‘<’) state=7;
                              else if(c==‘=’) state=11;
                                else if(c==‘>’) state=12;
                                    else return ERROR;
        s[i++]=c;
        break;
case 1:     if(IsLetter(c)||IsDigit(c)) state=1;
              else return ERROR;
            break;
case 2:     s[i]=0; GetBackChar();
            return IDENT;
case 3:     if(IsLetter(c)) state=4;
                    else if(IsDigit(c)) state=3;
                          else if(c==‘.’) state=5;
                             else return 4;
            s[i++]=c;       break;
case 4:     s[i]=0; GetBackChar();
            return INTEGER;
case 5:     if(IsDigit(c)) state=5;
            else state=6;
            s[i++]=0;
            break;
case 6:     s[i]=0; GetBackChar();
            return REAL;
case 7:     if(c==‘=’) state=8;
            else if(c==‘>’) state=9;
                 else state=10;
            s[i++]=c;
            break;
case 8:     s[i]=0;
            return LE;
case 9:     s[i]=0;
            return NE;
case 10:    s[i]=0; GetBackChar();
            return LE;
case 11:    s[i]=0;
```

```
                    return EQ;
            case 12:    if(c==‘=’) state=13;
                    else state=14;
                    s[i++]=c;
                    break;
            case 13:    s[i]=0;
                    return GE;
            case 14:    s[i]=0;
                    return GT;
        }
        if(c==0) break;
    }// end while
}// end function
```

Nhận xét:

Ưu điểm: chương trình dễ viết, trực quan đối với số lượng các loại từ tố là bé.

Nhược điểm: gặp nhiều khó khăn nếu số lượng loại từ tố là lớn, và khi cần bổ sung loại từ tố hoặc sửa đổi mẫu từ tố thì chúng ta lại phải viết lại chương trình.

5.2. Lập bộ phân tích từ vựng theo bảng chuyển

Để xây dựng chương trình bằng phương pháp này, điều cơ bản nhất là chúng ta phải xây dựng bảng chuyển trạng thái. Để tổng quát, thông tin của bảng chuyển trạng thái nên được lưu ở một file dữ liệu bên ngoài, như vậy sẽ thuận tiện cho việc chúng ta thay đổi dữ liệu chuyển trạng thái của ôtômát mà không cần quan tâm đến chương trình.

Đối với các trạng thái không phải là trạng thái kết thúc thì chúng ta chỉ cần tra bảng một cách tổng quát sẽ biết được trạng thái tiếp theo, và do đó chúng ta chỉ cần thực hiện các trường hợp cụ thể đối với các trạng thái kết thúc để biết từ tố cần trả về là gì.

Giả sử ta có hàm khởi tạo bảng trạng thái là: int InitStateTable();

Hàm phân loại ký hiệu đầu vào (ký hiệu kết thúc): int GetCharType();

Khi đó đoạn chương trình sẽ được mô tả như dưới đây:

```
#define STATE_NUM 100
#define TERMINAL _NUM 100
#define STATE_ERROR –1 // trạng thái lỗi
int table[STATE_NUM][TERMINAL_NUM]
```

```
// ban đầu gọi hàm khởi tạo bảng chuyển trạng thái.
InitStateTable();
int GetNextChar(); // hàm lấy ký tự tiếp theo
enum Token {IDENT, INTEGER, REAL, LT, LE, GT, GE, NE, EQ, ERROR};
// hàm này trả về loại từ tố
// từ vị nằm trong s
Token  GetNextToken(char *s)
{
  int state=0;
  int i=0;
  while(1)
  {       int c=GetNextChar();
          int type=GetCharType(c);
          switch(state)
          {
                  case 2:         s[i]=0; GetBackChar();
                          return IDENT;
case 4:           s[i]=0; GetBackChar();
                          return INTEGER;
                  case 6:                 s[i]=0; GetBackChar();
                          return REAL;
                  case 8:         s[i]=0;
                                  return LE;
                  case 9:         s[i]=0;
                                  return NE;
                  case 10:        s[i]=0; GetBackChar();
                                  return LE;
                  case 11:        s[i]=0;
                                  return EQ;
                  case 13:        s[i]=0;
                                  return GE;
                  case 14:        s[i]=0;
                                  return GT;
                  case STATE_ERROR: return ERROR;
                  defaulf:        state=table[state][type];
                                  s[i++]=c;
```

```
            }
            if(c==0) break;
      }// end while
    }// end function
```

Nhận xét:

Ưu điểm:

+ Thích hợp với bộ phân tích từ vựng có nhiều trạng thái, khi đó chương trình sẽ gọn hơn.

+ Khi cần cập nhật từ tố mới hoặc sửa đổi mẫu từ tố thì chúng ta chỉ cần thay đổi trên dữ liệu bên ngoài cho bảng chuyển trạng thái mà không cần phải sửa chương trình nguồn hoặc có sửa thì sẽ rất ít đối với các trạng thái kết thúc.

Nhược điểm: khó khăn cho việc lập bảng, kích thước bảng nhiều khi là quá lớn, và không trực quan.

6. XÁC ĐỊNH LỖI TRONG PHÂN TÍCH TỪ VỰNG

Chỉ có rất ít lỗi được phát hiện trong lúc phân tích từ vựng, vì bộ phân tích từ vựng chỉ quan sát chương trình nguồn một cách cục bộ, không xét quan hệ cấu trúc của các từ với nhau.

Ví dụ: khi bộ phân tích từ vựng gặp xâu fi trong biểu thức

fi a= b then . . .

thì bộ phân tích từ vựng không thể cho biết rằng fi là từ viết sai của từ khoá if hoặc là một tên không khai báo. Nó sẽ nghiễm nhiên coi rằng fi là một tên đúng và trả về một từ tố tên. Chú ý lỗi này chỉ được phát hiện bởi bộ phân tích cú pháp.

Các lỗi mà bộ phân tích từ vựng phát hiện được là các lỗi về một từ vị không thuộc một loại từ tố nào, *ví dụ như gặp từ vị 12xyz.*

Bộ xử lý lỗi phải đạt mục đích sau:

- Thông báo lỗi một cách rõ ràng và chính xác.
- Phục hồi lỗi một cách nhanh chóng để xác định lỗi tiếp theo.
- Không làm chậm tiến trình của một chương trình đúng.

Khi gặp những lỗi có 2 cách xử lý:

+ Hệ thống sẽ ngừng hoạt động và báo lỗi cho người sử dụng.

+ Bộ phân tích từ vựng ghi lại các lỗi và cố gắng bỏ qua chúng để hệ thống tiếp tục làm việc, nhằm phát hiện đồng thời thêm nhiều lỗi khác. Mặt khác, nó còn có thể tự sửa (hoặc cho những gợi ý cho những từ đúng đối với từ bị lỗi).

Cách khắc phục là:

- Xoá hoặc nhảy qua kí tự mà bộ phân tích từ vựng không tìm thấy từ tố (panic mode).

- Thêm kí tự bị thiếu.
- Thay một kí tự sai thành kí tự đúng.
- Tráo 2 kí tự đứng cạnh nhau.

BÀI TẬP

1. Phân tích các chương trình pascal và C sau thành các từ tố và thuộc tính tương ứng.

a) pascal:

```
Function max(i,j:integer): Integer; {Trả lại số lon nhất trong 2 số nguyên i, j }
Begin
  If i>j then max:=i;
  Else max:=j;
End;
```

B) C:

```
Int max(int i, int j)
/* Trả lại số lon nhất trong 2 số nguyên i, j*/
{return i>j?i:j;}
```

Hãy cho biết có bao nhiêu từ tố được đưa ra và chia thành bao nhiêu loại?

2. Phân tích các chương trình pascal và c sau thành các từ tố và thuộc tính tương ứng.

a) pascal

```
var i,j;
begin
        for i= 0 to 100 do j=i;
                write('i=', 'j:=',j);
end;
```

B) C:

```
Int i,j:
        Main(void
        {
          for (i=0; i=100;i++)
                printf("i=%d;",i,"j=%d",j= =i);
        }
```

3. Mô tả các ngôn ngữ chỉ định bởi các biểu thức chính quy sau:

a. 0(0|1)*0 b.((ε|0)1*)*

4. Viết biểu thức chính quy cho: tên, số nguyên, số thực, char, string… trong pascal. Xây dựng đồ thị chuyển cho chúng. Sau đó, kết hợp chúng thành đồ thị chuyển duy nhất.

5. Dựng đồ thị chuyển cho các mô tả dưới đây.

a. Tất cả các xâu chữ cái có 6 nguyên âm a, e, i, o, u, y theo thứ tự. Ví dụ: “abeiptowwrungfhy”

b. tất cả các xâu số không có một số nào bị lặp.

c. tất cả các xâu số có ít nhất một số nào bị lặp.

d. tất cả các xâu gồm 0,1, không chứa xâu con 011.

Bài tập thực hành

Bài 1: Xây dựng bộ phân tích từ vựng cho ngôn ngữ pascal chuẩn.

Bài 2: Xây dựng bộ phân tích từ vựng cho ngôn ngữ C chuẩn.

CHƯƠNG 3

PHÂN TÍCH CÚ PHÁP VÀ CÁC PHƯƠNG PHÁP PHÂN TÍCH CƠ BẢN

Mục tiêu: *Sinh viên cần nắm được:*

- *Các phương pháp phân tích cú pháp và các chiến lược phục hồi lỗi trong quá trình phân tích cú pháp.*
- *Cách tự cài đặt một bộ phân tích cú pháp từ một văn phạm phi ngữ cảnh xác định*

Nội dung: *Nghiên cứu các phương pháp phân tích:*

- Phương pháp phân tích cơ bản (phương pháp phân tích quay lui: topdown, bottom – up)

- Phương pháp phân tích hiệu quả (phương pháp phân tích tất định: LL, LR).

1. SƠ LƯỢC VỀ VĂN PHẠM PHI NGỮ CẢNH

1.1 Định nghĩa

* Dạng BNF (Backus – Naur Form) của văn phạm phi ngữ cảnh

+ Các ký tự viết hoa: biểu diễn ký hiệu không kết thúc, (có thể thay bằng một xâu đặt trong dấu ngoặc < > hoặc một từ in nghiêng).

+ Các ký tự viết chữ nhỏ và dấu toán học: biểu diễn các ký hiệu kết thúc (có thể thay bằng một xâu đặt trong cặp dấu nháy kép " " hoặc một từ in đậm).

+ Ký hiệu -> hoặc = là: ký hiệu chỉ phạm trù cú pháp ở vế trái được giải thích bởi vế phải.

+ Ký hiệu | chỉ sự lựa chọn.

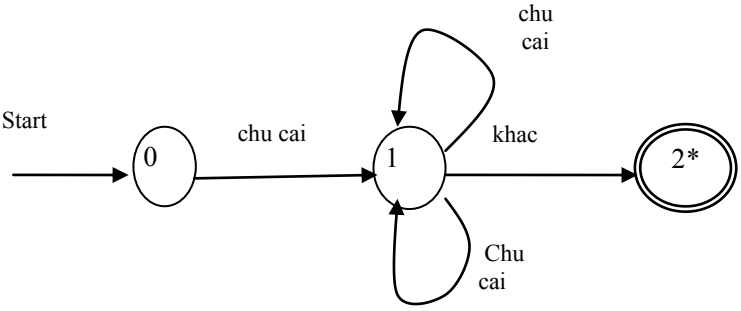
Ví dụ: <Toán hạng> = <Tên> | <Số> | "(" <Biểu thức> ")"

hoặc ToánHạng -> Tên | Số | (BiểuThức

1.2 Đồ thị chuyển biểu diễn văn phạm phi ngữ cảnh

- Các vòng tròn với ký hiệu bên trong biểu thị cho trạng thái. Các chữ trên các cung biểu thị cho ký hiệu vào tiếp theo. Trạng thái vẽ bằng một vòng tròn kép là trạng thái kết thúc.

Nếu trạng thái kết thúc có dấu * nghĩa là ký hiệu cuối không thuộc xâu đoán nhận.



Hình 3.1: Đồ thị chuyển cho từ tố Tên

1.3 Cây suy dẫn

* <u>*Suy dẫn*</u> *Cho văn phạm G=(T,N,P,S)*

- Suy dẫn trực tiếp là một quan hệ hai ngôi ký hiệu => trên tập V* nếu αβγ là một xâu thuộc V* và β->δ là một sản xuất trong P, thì αβγ => αδγ.

- Suy dẫn k bước, ký hiệu là $\overset{k}{=>}$ hay $\alpha \overset{k}{=>} \beta$ nếu tồn tại dãy $\alpha_0, \alpha_1, \ldots, \alpha_k$ sao cho: $\alpha = \alpha_0 => \alpha_1 => \ldots => \alpha_k = \beta$

- Xâu α suy dẫn xâu nếu k>=0 và ký hiệu là $\alpha \overset{*}{=>} \beta$

- Xâu α suy dẫn không tầm thường xâu β nếu k>0 và ký hiệu là $\alpha \overset{+}{=>} \beta$

* <u>*Cây phân tích*</u> (cây suy dẫn)

Cây phân tích trong một văn phạm phi ngữ cảnh G = (T,N,P,S) là một cây thỏa mãn các điều kiện sau:

1. Mọi nút có một nhãn, là một ký hiệu trong $(T \cup N \cup \{\varepsilon\})$
2. Nhãn của gốc là S
3. Nếu một nút có nhãn X là một nút trong thì $X \in N$
4. Nếu nút n có nhãn X và các nút con của nó theo thứ tự trái qua phải có nhãn $Y_1, Y_2, \ldots, Y_k$ thì $X->Y_1Y_2 \ldots Y_k$ sẽ là một sản xuất $\in P$
5. Nút lá có nhãn thuộc T hoặc là ε

* <u>*Suy dẫn trái nhất*</u> (suy dẫn trái), nếu ở mỗi bước suy dẫn, biến được thay thế là biến nằm bên trái nhất trong dạng câu.

* <u>*Suy dẫn phải nhất*</u> (suy dẫn phải), nếu ở mỗi bước suy dẫn, biến được thay thế là biến nằm bên phải nhất trong dạng câu.

1.4 Văn phạm nhập nhằng(mơ hồ)

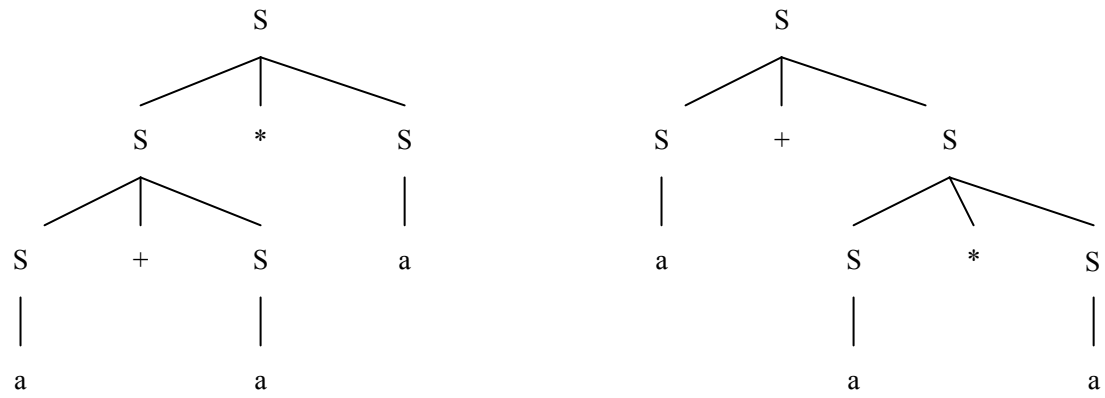
Một văn phạm G được gọi là văn phạm nhập nhằng nếu có một xâu α là kết quả của hai cây suy dẫn khác nhau trong G. Ngôn ngữ do văn phạm này sinh ra gọi là ngôn ngữ nhập nhằng.

Ví dụ:

Xét văn phạm G cho bởi các sản xuất sau: S -> S + S | S * S | (S) | a

Với xâu vào là w = “a+a*a” ta có:

Văn phạm này là nhập nhằng vì có hai cây đối với câu vào w như sau:



Chúng ta có ví dụ đối với suy dẫn trái (đối với cây đầu tiên) là:

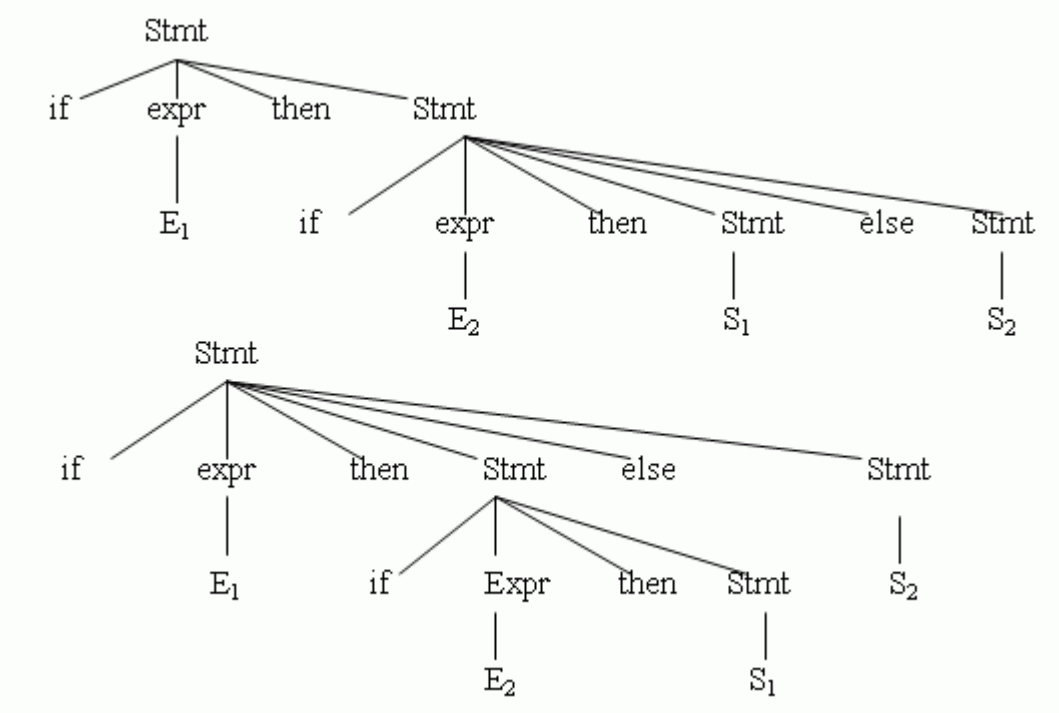
S => S * S => S + S => S + S * S => a + S * S => a + a * S => a + a * a

Suy dẫn phải (đối với cây đầu tiên) là:
S => S * S => S * a => S + S * a => S + a * a => a + a * a

* Đôi khi có thể viết lại một văn phạm nhằm tránh sự mơ hồ của nó. Một ví dụ, chúng ta sẽ loại bỏ sự mơ hồ trong văn phạm sau :

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

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



Hình 3.2 Hai cây phân tích cú pháp cho một câu nhập

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

Stmt → matched_stmt | unmatched_stmt

matched_stmt → **if** expr **then** matched_stmt **else** matched_stmt

| **other**

unmatched_stmt → **if** expr **then** Stmt

if expr **then** matched_stmt **else** unmatched_stmt

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

1.5 Đệ qui

* *Định nghĩa*: Ký hiệu không kết thúc A của văn phạm gọi là đệ qui nếu tồn tại:

$A \Rightarrow^{+} \alpha A \beta$ với $\alpha, \beta \in V^{+}$

- Nếu $\alpha = \varepsilon$ thì A gọi là đệ qui trái
- Nếu $\beta = \varepsilon$ thì A gọi là đệ qui phải
- Nếu $\alpha, \beta \notin \varepsilon$ thì A gọi là đệ qui trong

* *Có 2 loại đệ quy trái:*

- ✓ Loại ttrực tiếp: có dạng $A \rightarrow A\alpha$ ($A \Rightarrow^{+} A\alpha$)
- ✓ Loại gián tiếp: Gây ra do nhiều bước suy dẫn.

Ví dụ: $S \rightarrow Aa \mid b$; $A \rightarrow Ac \mid Sd$;

S là đệ qui trái vì $S \Rightarrow Aa \Rightarrow Sda$

* *Loại bỏ đệ qui trái* (loại bỏ suy dẫn $A \Rightarrow^{+} A\alpha$)

- Giả sử có luật đệ qui trái $A \rightarrow A\alpha \mid \beta$ chúng ta thay các luật này bằng các luật:

$$A \rightarrow \beta A' \quad \text{và} \quad A' \rightarrow \alpha A' \mid \varepsilon$$

- Tổng quát hoá lên ta có:

Nếu có các luật đệ qui trái: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

trong đó không β_i nào bắt đầu bằng A. Thay các sản xuất này bởi các sản xuất:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$ và $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Ví dụ2: Xét văn phạm biểu thức số học sau:

$\{E \rightarrow E + T \mid T;\quad T \rightarrow T * F \mid F;\quad F \rightarrow (E) \mid a\}$

Loại bỏ đệ qui trái trực tiếp cho các sản xuất của E rồi của T, ta được văn phạm mới không còn sản xuất có đệ qui trái như sau:

{E → TE'; E' → +TE' | ε; T → FT'; T' → *FT' | ε; F → (E) | a}

Qui tắc này loại bỏ được đệ qui trái trực tiếp nằm trong các sản xuất nhưng không loại bỏ được đệ qui trái nằm trong các dẫn xuất có hai hoặc nhiều bước. Qui tắc này cũng không loại bỏ được đệ qui trái ra khỏi sản xuất A->A.

Với đệ quy trái và đệ quy gián tiếp ta có thể dùng giải thuật sau để loại bỏ:

Input: Văn phạm không không có dạng $A \Rightarrow^{+} A$ hoặc $A \Rightarrow^{+} \varepsilon$
Output: Văn phạm tương đương không đệ qui trái
Phương pháp:
1. Sắp xếp các ký hiệu không kết thúc theo thứ tự $A_1, A_2 .. .. A_n$
2. For i:= 1 to n do
  Begin
    for j:=1 to i-1 do
      Begin
        if (∃ đệ quy trái gián tiếp của A_j qua A_i) then
          Thay sản xuất $A_i \rightarrow A_j\alpha \in P$ bởi sản xuất $A_i \rightarrow \gamma_1\alpha \mid \gamma_2\alpha \mid \ldots \mid \gamma_n\alpha$
            Trong đó $A_j \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \in P$
      End;
    Loại bỏ đệ qui trái trực tiếp cho A_i
  End;

Ví dụ: G = { S → Aa | b; A→Ac | Sd}

Sắp xếp các ký hiệu chưa kết thúc theo thứ tự S,A..

Với i=1, không có đệ qui trái trực tiếp nên không có điều gì xảy ra.

Với i=2 , thay luật sinh A→ Sd được A → Aad | bd.

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

S → Aa |b; A → bdA'; A' → cA' | adA' | ε

1.6 Phép thừa số hoá trái

Thừa số hoá trái (left factoring) là một phép biến đổi văn phạm nhằm sinh ra một văn phạm thích hợp cho việc phân tích cú pháp không quay lui. Ý tưởng cơ bản là khi không rõ sản xuất nào trong trong hai sản xuất có cùng vế trái là A được dùng để khai triển A thì ta có thể viết lại các sản xuất này nhằm "hoãn lại quyết định", cho đến khi có đủ thông tin để đưa ra được quyết định lựa chọn sản xuất nào.

- Nếu có hai sản xuất A -> $\alpha\beta_1$ | $\alpha\beta_2$ thì ta không biết phải khai triển A theo $\alpha\beta_1$ hay $\alpha\beta_2$. Khi đó, thay hai sản xuất này bằng: A -> αA'; A' -> β_1 | β_2

Ví dụ: S -> iEtS | iEtSeS | a; E -> b

Khi được thừa số hoá trái, văn phạm này trở thành:

S -> iEtSS' | a; S' -> eS | ε; E -> b

Vì thế khi cần khai triển S với ký hiệu xâu vào hiện tại là i, chúng ta có thể lựa chọn iEtSS' mà không phải băn khoăn giữa iEtS và iEtSeS của văn phạm cũ.

Giải thuật tạo thừa số hoá trái cho văn phạm:

Input: Văn phạm G

Output: Văn phạm tương đương với nhân tố trái.

Phương pháp:

Với mỗi ký hiệu chưa kết thúc A, có các ký hiệu dẫn đầu các vế phải giống nhau, ta tìm một chuỗi a là chuỗi có độ dài lớn nhất chung cho tất cả các vế phải (a là nhân tố trái)

Giả sử A $\rightarrow ab_1| ab_2|.. .. | ab_n | g$

Trong đó g không có chuỗi dẫn đầu chung với các vế phải khác. Biến đổi luật sinh thành: $A \rightarrow a A' | g$; $A' \rightarrow b_1| b_2 | .. .. | b_n$

2. CÁC PHƯƠNG PHÁP PHÂN TÍCH

- Mọi ngôn ngữ lập trình đều có các luật mô tả các cấu trúc cú pháp. Một chương trình viết đúng phải tuân theo các luật mô tả này. Phân tích cú pháp là để tìm ra cấu trúc dựa trên văn phạm của một chương trình nguồn.

- Thông thường có hai chiến lược phân tích*:*

+ Phân tích trên xuống (topdown): Cho một văn phạm PNC G = (Σ, Δ, P, S) và một câu cần phân tích w. Xuất phát từ S áp dụng các suy dẫn trái, tiến từ trái qua phải thử tạo ra câu w.

+ Phân tích dưới lên (bottom-up): Cho một văn phạm PNC G = (Σ, Δ, P, S) và một câu cần phân tích w. Xuất phát từ câu w áp dụng thu gọn các suy dẫn phải, tiến hành từ trái qua phải để đi tới kí hiệu đầu S.

Theo cách này thì phân tích Topdown và LL(k) là phân tích trên xuống, phân tích Bottom-up và phân tích LR(k) là phân tích dưới lên.

* *Điều kiện để thuật toán dừng:*

+ Phân tích trên xuống dừng khi và chỉ khi G kông có đệ quy trái.

+ Phân tích dưới lên dừng khi G không chứa sản xuất $A \Rightarrow^+ A$ và $A \rightarrow \varepsilon$.

2.1 Phân tích topdown

** Thuật toán phân tích Top-down*

Input: Văn phạm PNC G = (Σ, Δ, P, S) không đệ quy trái, xâu w = $a_1, a_2, \ldots a_n$

Output: Cây phân tích từ trên xuống của xâu w (w ∈ L(G)), báo lỗi (w ∉ L(G)).

Method:

Dùng một con trỏ chỉ đến xâu vào w. Ký hiệu trên xâu vào do con trỏ chỉ đến gọi là ký hiệu vào hiện tại.

1) Khởi tạo cây với gốc là S, con trỏ trỏ đến kí hiệu đầu tiên của xâu w là a_1.

2) Kiểm tra nút đang xét:

2.1. Nếu nút đang xét là biến A thì chọn một sản xuất có vế trái là A trong P (giả sử sản xuất $A \rightarrow X_1...X_k$) để triển khai. Lấy nút X_1 làm nút đang xét. *(các Xi ∈ Σ ∪ Δ ∪{ε})*

2.2. Nếu nút đang xét là kí hiệu kết thúc a ∈ Σ thì đối sánh a với ký hiệu vào hiện tại a_1

+ a = a_1: lấy nút ngay bên phải a làm nút đang xét, con trỏ dịch sang bên phải một ký hiệu trên xâu w.

+ a khác a_1: quay lại nút trước đó và lặp lại b2 với thử lựa chọn sản xuất khác.

2.3. Nếu nút đang xét là ε thì chuyển sang nút ngay bên phải nó làm nút đang xét

Tiếp tục lặp lại bước 2. Thủ tục trên lặp lại sau hữu hạn bước và có 2 khả năng xảy ra:

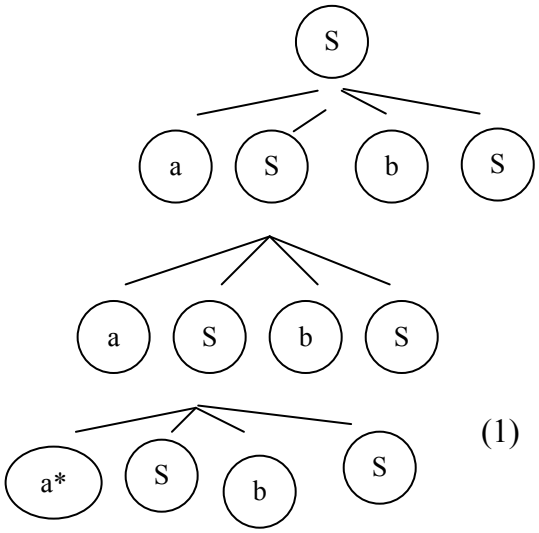
+ Nếu đối sánh hết xâu vào và cây không còn nút nào chưa xét nữa thì được một cây phân tích.

+ Nếu đã quay lui hết các trường hợp mà không sinh được cây phân tích thì kết luận w ∉ L(G)

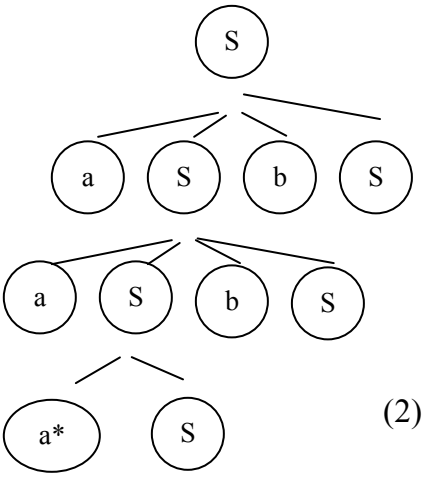
* Điều kiện để một văn phạm phi ngữ cảnh phân tích được bởi thuật toán Top-down là văn phạm không có đệ qui trái.

* Độ phức tạp thuật toán là hàm số mũ n với n là độ dài xâu vào.

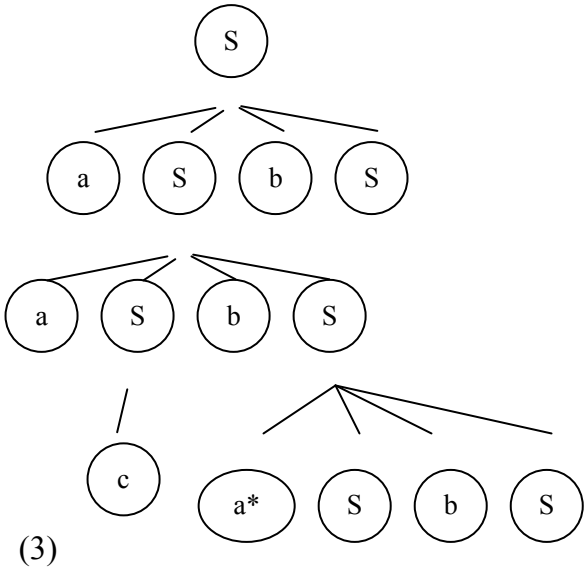
Ví dụ: Cho văn phạm G = {S→ aSbS | aS | c} Phân tích xâu **"aacbc"** bằng thuật toán Top-down, vẽ cây phân tích trong quá trình phân tích quay lui.


S
a
S
b
S
a
S
b
S
a*
S
b
S

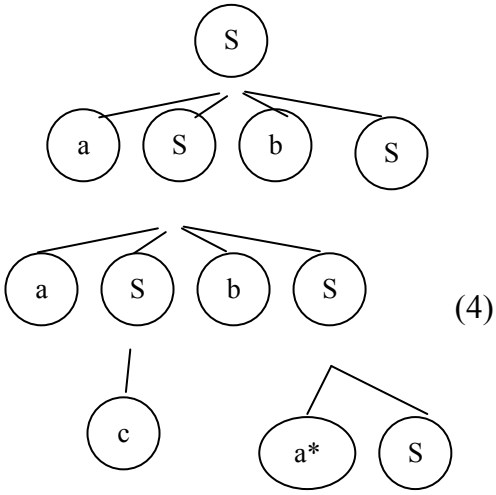
(1)


S
a
S
b
S
a
S
b
S
a*
S

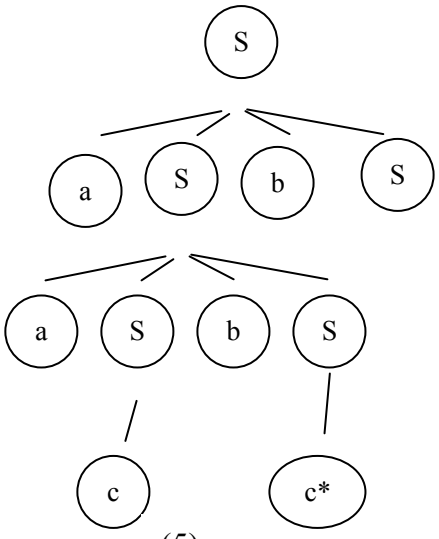
(2)


S
a
S
b
S
a
S
b
S
c
a*
S
b
S

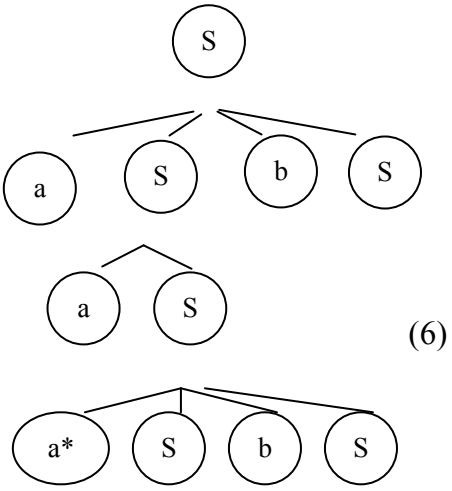
(3)


S
a
S
b
S
a
S
b
S
c
a*
S

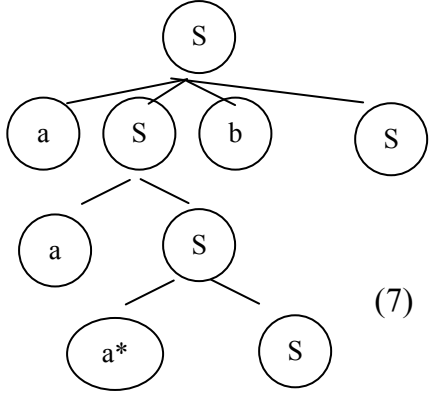
(4)


S
a
S
b
S
a
S
b
S
c
c*

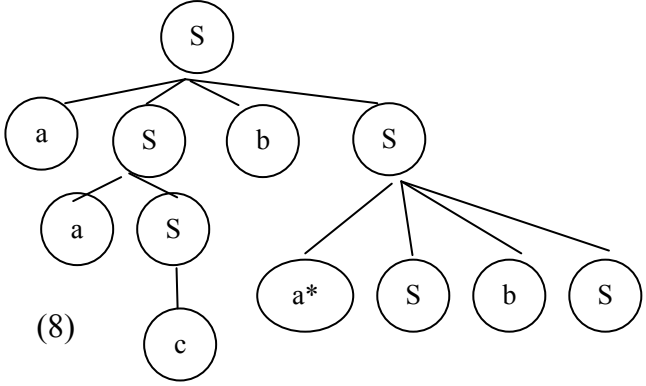
(5)


S
a
S
b
S
a
S
a*
S
b
S

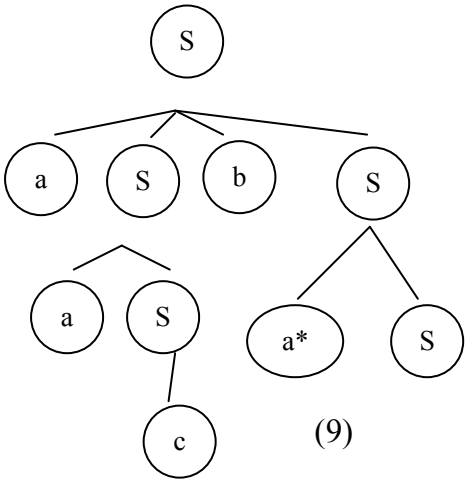
(6)



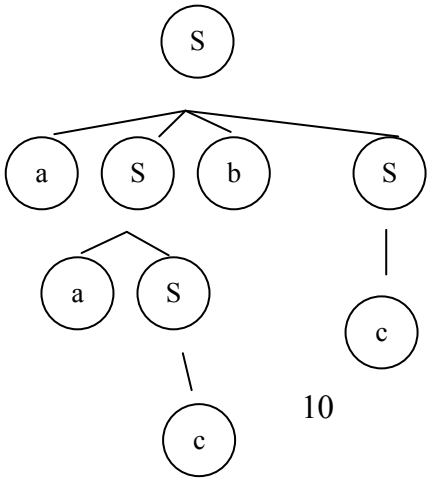
(7)



(8)



(9)



10

2.2 Phân ttích bottom - up

Trong phương pháp này, ta xây dựng cú pháp cho xâu nhập bắt đầu từ lá lên tới gốc. Đây là quá trình rút gọn một xâu thành một kí hiệu mở đầu của văn phạm. Tại mỗi bước rút gọn, một xâu con bằng một xâu phải của một sản xuất nào đó thì xâu con này được thay thế bởi vế trái của sản xuất đó. (còn gọi là phương pháp gạt thu gọn - shift reduce parsing).

* *Cấu tạo:* - 1 STACK: Lưu kí hiệu văn phạm, đáy là $

- 1 BUFFER INPUT chứa chuỗi w, cuối xâu them kí hiệu $.

* *Hoạt động:*

- Khởi tạo: stack: $, inputbuffer chứa w$.

- Bộ phận tích gạt lần lượt các ký hiệu đầu vào từ trái sang phải vào ngăn xếp đến khi nào đạt được một thu gọn (các kí tự trên đỉnh ngăn sếp là vế phải của một sản xuất $\in$ P) thì thu gọn (thay thế vế phải xuất hiện trên đỉnh ngăn xếp bởi vế trái của sản xuất đó). Nếu có nhiều cách thu gọn tại một trạng thái thì lưu lại cho quá trình quay lui. Quá trình cứ tiếp tục, nếu dừng lại mà chưa đạt đến trạng thái kết thúc thì quay lại bước quay lui gần nhất và thử với lựa chọn sản xuất khác hoặc hành động khác.

Quá trình phân tích lặp sau hữu hạn bước.

- Nếu quá trình đạt đến trạng thái kết thúc (trạng thái ngăn xếp chỉ chứa $S và xâu vào chỉ chứa $) thì quá trình kết thúc và phân tích thành công.

- Nếu đã xét hết tất cả các trường hợp (không quay lui được nữa) mà chưa đạt đến trạng thái kết thúc thì dừng lại và thông báo xâu vào không phân tích được bởi văn phạm đã cho.

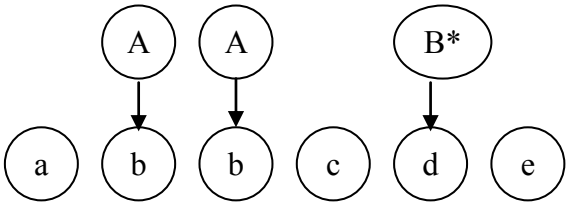
Ví dụ: Cho văn phạm G = {S → aABe; A → Abc | b; B → d}

Phân tích câu vào "abbcde"

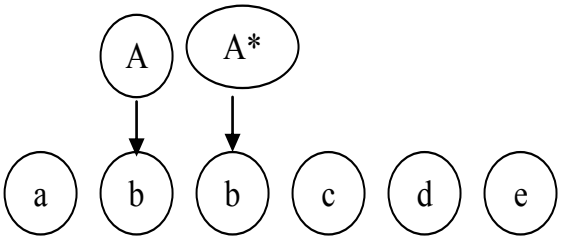
Quá trình phân tích Bottom-up như sau:

Ngăn xếp	**Đầu vào**	**Hành động**
$	abbcde$	gạt
$a	bbcde$	gạt
$ab	bcde$	thu gọn A -> b
$aA	bcde$	gạt
$aAb	cde$	thu gọn A -> b (2)
$aAA	cde$	gạt
$aAAc	de$	gạt
$aAAcd	e$	thu gọn B -> d (1)
$aAAcB	e$	gạt
$aAAcBe	$	dừng, quay lui 1 (gạt)
$aAAcde	$	dừng, quay lui 2 (gạt)
$aAbc	de$	thu gọn A -> Abc
$aA	de$	gạt
$aAd	e$	thu gọn B -> d
$aAB	e$	gạt
$aABe	$	thu gọn S -> aABe
$S	$	**Chấp nhận**

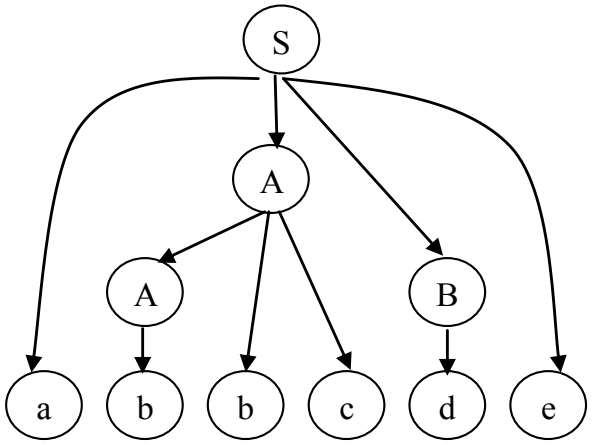
Vẽ cây cho quá trình phân tích và quay lui trên:



Quá trình 1



Quá trình 2



Quá trình 3

Quá trình suy dẫn: Abbcde => aAbcde => aAde => aABe => S

Viết ngược lại, ta được dẫn xuất phải nhất:

$S =>_{rm} aABe =>_{rm} aAde =>_{rm} aAbcde =>_{rm} abbcde$

Phân tích Bottom-up không phân tích được văn phạm có các sản xuất

$B \rightarrow \varepsilon$ hoặc có suy dẫn $A =>^{+} A$.

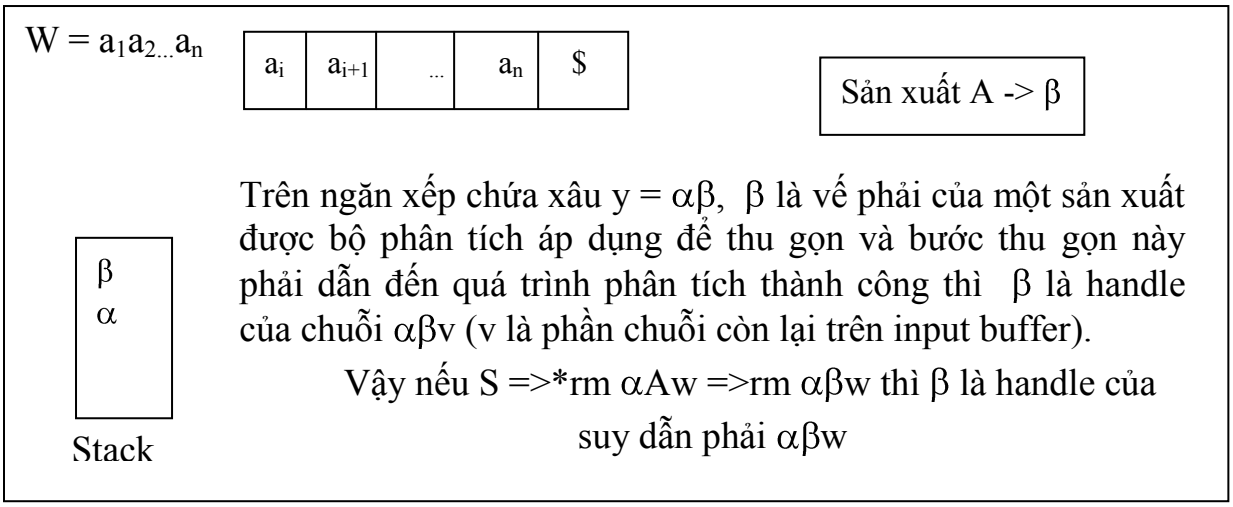
* ***Handle của một chuỗi:*** là chuỗi con của nó và là vế phải của một sản xuất trong phép thu gọn nó thành ký hiệu vế trái của 1 sản xuất.

Ví dụ: Trong ví dụ trên.

Ngăn xếp	**Đầu vào**	**Hành động**	**Handle**	**Suy dẫn phải**	**Tiền tố khả tồn**
$	abbcde$	gạt			
$a	bbcde$	gạt		abbcde	a
$ab	bcde$	thu gọn A -> b	b	abbcde	ab
$aA	bcde$	gạt		aAbcde	aA
$aAb	cde$	thu gọn A -> b (2)	b	aAbcde	aAb
$aAA	cde$	gạt			
$aAAc	de$	gạt			
$aAAcd	e$	thu gọn B -> d (1)	d không phải là handle do áp dụng thu gọn này là không thành công		
$aAAcB	e$	gạt			
$aAAcBe	$	dừng, quay lui 1 (gạt)			

$aAAcde	$	dừng, quay lui 2 (gạt)			
$aAbc	de$	thu gọn A -> Abc	Abc	AAbcde	
$aA	de$	gạt			
$aAd	e$	thu gọn B -> d	d	AAde	
$aAB	e$	gạt			
$aABe	$	thu gọn S -> aABe			
$S	$	**chấp nhận**			

Chú ý Handle là chuỗi mà chuỗi đó phải là một kết quả của suy dẫn phải từ S và phép thu gọn xảy ra trong suy dẫn đó.



* ***<u>Tiền tố khả tồn</u> (viable prefixes)***

Xâu ký hiệu trong ngăn xếp tại mỗi thời điểm của một quá trình phân tích gạt - thu gọn là một tiền tố khả tồn.

Ví dụ: tại một thời điểm trong ngăn xếp có dữ liệu là αβ và xâu vào còn lại là w thì αβw là một dạng câu dẫn phải và αβ là một tiền tố khả tồn.

2.3 Phân tích LL

Tử tưởng của phương pháp phân tích LL là khi ta triển khai một ký hiệu không kết thúc, lựa chọn cẩn thận các sản xuất như thế nào đó để tránh việc quay lui mất thời gian. Tức là phải có một cách nào đó xác định dực ngay lựa chọn đúng mà không phải thử các lựa chọn khác. Thông tin để xác định lựa chọn dựa vào những gì đã biết trạng thái và kí hiệu kết thúc hiện tại.

LL: là một trong các phương pháp phân tích hiệu quả, thuộc chiến lược phân tích topdown nhưng hiệu quả ở chỗ nó là phương pháp phân tích không quay lui.

- Bộ phân tích tất định: Các thuật toán phân tích có đặc điểm chung là xâu vào được quét từ trái sang phải và quá trình phân tích là hoàn toàn xác định, do đó ta

gọi là bộ phân tích tất định. (Phân tích topdown và bottom – up có phải là phân tích tất định không? – không do quá trình phân tích là không xác định).

L: left – to – right (quét từ phải qua trái) L : leftmosst – derivation (suy dẫn trái nhất); k là số ký hiệu nhìn trước để đưa ra quyết định phân tích.

Giả sử ký hiệu không kết thúc A có các sản xuất: A -> α_1 | α_2 | . . . | α_n thoả mãn tính chấ:t các xâu α_1, α_2, . . ., α_n suy dẫn ra các xâu với ký hiệu tại vị trí đầu tiên là các ký hiệu kết thúc khác nhau, khi đó chúng ta chỉ cần nhìn vào ký hiệu đầu vào tiếp theo sẽ xác định được cần khai triển A theo α_i nào. Nếu cần tới k ký hiệu đầu tiên thì mới phân biệt được các xâu α_1, α_2, . . ., α_n thì khi đó để chọn luật sản xuất nào cho khai triển A ta cần nhìn k ký hiệu đầu vào tiếp theo.

Văn phạm LL(k) là văn phạm cho phép xây dựng bộ phân tích làm việc tất định nếu bộ phân tích này được phép nhìn k kí hiệu vào nằm ngay bên phải của vị trí vào hiện tại.

Ngôn ngữ sinh ra bởi văn phạm LL(k) là ngôn ngữ LL(k). Thông thường chúng ta xét với k=1.

2.3.1 First và follow

*** First của một xâu:**

First(α) cho chúng ta biết xâu α có thể suy dẫn đến tận cùng thành một xâu bắt đầu bằng ký hiệu kết thúc nào.

Định nghĩa First(α) là tập chứa tất cả các ký hiệu kết thúc a mà a có thể là bắt đầu của một xâu được suy dẫn từ α

$$+ \text{First}(\alpha) = \{a \in T \mid \alpha =>^* a\beta \}$$

$$+ \varepsilon \in \text{First}(\alpha) \text{ nếu } \alpha =>^* \varepsilon$$

Thuật toán tính First(X) với X là một ký hiệu văn phạm

1. nếu X là ký hiệu kết thúc thì First(X) = {X}
2. nếu X -> ε là một sản xuất thì thêm ε vào First(X)
3. nếu X -> $Y_1...Y_k$ là một sản xuất thì thêm First(Y_1) vào First(X) trừ ε

   nếu First(Y_t) chứa ε với mọi t=1,...,i với i<k thì thêm First(Y_{i+1}) vào First(X) trừ ε. Nếu trường hợp i=k thì thêm ε vào First(X)

Cách tính First(α) với α là một xâu

Giả sử α= X_1X_2 . . . X_k. Ta tính như bước 3 của thuật toán trên:

1. thêm First(X_1) vào First(α) trừ ε

2. nếu First(X_t) chứa ε với mọi t=1,...,i với i<k thì thêm First(X_{i+1}) vào First(α) trừ ε. Nếu trường hợp i=k thì thêm ε vào First(α)

- Tính First của các ký hiệu không kết thúc: lần lượt xét tất cả các sản xuất.Tại mỗi sản xuất, áp dụng các qui tắc trong thuật toán tính First để thêm các ký hiệu vào các tập First. Lặp lại và dừng khi nào gặp một lượt duyệt mà không bổ sung thêm được bất kỳ ký hiệu nào vào tập First và ta đã tính xong các tập First cho các ký hiệu.

Ví dụ 1: Cho văn phạm sau: S -> AB; A -> aA | ε; B -> bB | ε

Hãy tính First của các ký hiệu S, A, B

Kết quả: Fisrt(A) = {a, ε}; First(B) = {b,ε}; First(S) = {a,b,ε}

*** Follow của một ký hiệu không kết thúc:**

Định nghĩa follow(A) A là kí hiệu không kết thúc.

Follow(A) với A là ký hiệu không kết thúc là tập các ký hiệu kết thúc a mà chúng có thể xuất hiện ngay bên phải của A trong một số dạng câu. Nếu A là ký hiệu bên phải nhất trong một số dạng câu thì thêm $ vào Follow(A).

+ Follow(A) = {a∈T | ∃ S =>* αAaβ }

+ $ ∈ Follow(A) khi và chỉ khi tồn tại suy dẫn S =>* αA

Thuật toán tính Follow(A) với A là một ký hiệu không kết thúc

1. Thêm $ vào Follow(S) với S là ký hiệu bắt đầu (chú ý là nếu ta xét một tập con với một ký hiệu E nào đó làm ký hiệu bắt đầu thì cũng thêm $ vào Follow(E)).
2. Nếu có một sản xuất dạng B->αAβ và β≠ε thì thêm các phần tử trong First(β) trừ ε vào Follow(A).

*Thật vậy: nếu a ∈ First(β) thì tồn tại β=>*aγ, khi đó, do có luật B->αAβ nên tồn tại S =>* $\alpha_1 B\beta_1$ => $\alpha_1\alpha A\beta\beta_1 => \alpha_1\alpha Aa\gamma\beta_1$ Theo định nghĩa của Follow thì ta có a ∈ Follow(A)*

3. Nếu có một sản xuất dạng B->αA hoặc B->αAβ với ε∈First(B) thì mọi phần tử thuộc Follow(B) cũng thuộc Follow(A)

Thật vậy: nếu a ∈ Follow(B) thì theo định nghĩa Follow ta có S => $\alpha_1 Ba\beta_1$ =>* $\alpha 1\alpha Aa\beta_1$, suy ra a ∈ Follow(A)*

- Để tính Follow của các ký hiệu không kết thúc: ta xét các sản xuất. Tại mỗi sản xuất, áp dụng qui tắc tính Follow để thêm các ký hiệu vào tập Follow. Lặp lại và dừng khi không bổ sung được ký hiệu nào vào các tập Follow.

Ví dụ ở trên, ta tính được tập Follow cho các ký hiệu S, A, B như sau:

Follow(S) = {$} Follow(A) = {b,$} Follow(B) = {$}

2.3.2 Lập bảng phân tích LL(1)

Bảng phân tích LL(1) là một mảng hai chiều: Một chiều chứa các ký hiệu không kết thúc, chiều còn lại chứa các ký hiệu kết thúc và $.

Vị trí M(A,a) chứa sản xuất $A \rightarrow \alpha$ trong bảng chỉ dẫn cho ta biết rằng khi cần khai triển ký hiệu không kết thúc A với ký hiệu đầu vào hiện tại là a thì áp dụng sản xuất $A \rightarrow \alpha$.

Thuật toán xây dựng bảng LL(1):

Input: Văn phạm G.

Output: Bảng phân tích M.

Phương pháp:

Với mỗi sản xuất A->α, thực hiện bước 2 và bước 3

1. Với mỗi ký hiệu kết thúc a ∈ First(α), định nghĩa mục M(A,a) là A->α
2. Nếu mỗi b ∈ Follow(A) thì định nghĩa mục M(A,b) là A->α (nếu ε ∈ First(α) và $ ∈ Follow(A) thì thêm A->α vào M[A,$])

Đặt tất cả các vị trí chưa được định nghĩa trong bảng là "lỗi".

Cho văn phạm sau: $S \rightarrow AB;\quad A \rightarrow aA \mid \varepsilon;\quad B \rightarrow bB \mid \varepsilon$

Lập bảng phân tích:

Kí hiệu	**First**	**Follow**
S	a, b, ε	$
A	a, ε	$, b
B	b, ε	$

Kí tự chưa kết thúc	Kí tự kết thúc		
	a	b	$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow aA$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B	$B \rightarrow bB$		$B \rightarrow \varepsilon$

2.3.3 Văn phạm LL (k) và LL (1)

Giải thuật trên có thể áp dụng bất kỳ văn phạm G nào để sinh ra bảng phân tích M. Tuy nhiên có những văn phạm (đệ quy trái và nhập nhằng) thì trong bảng phân tích M có những ô chứa nhiều hơn một luật sinh.

Ví dụ: Văn phạm {S → iEtSS'| a S' → eS | ε E → b }

KÍ TỰ CHƯA KẾT	***Kí tự kết thúc***					
	a	b	e	i	t	$
S	S → a			S→ iEtSS'		
S'			S → ε S' → eS			S'→ ε
E		E→ b				

* Định nghĩa: Văn phạm LL(1) là văn phạm xây dựng được bảng phân tích M có các ô chỉ được định nghĩa nhiều nhất là một lần.

** Điều kiện để một văn phạm là LL(1)*

- Để kiểm tra văn phạm có phải là văn phạm LL(1) hay không ta lập bảng phân tích LL(1) cho văn phạm đó. Nếu có mục nào trong bảng được định nghĩa nhiều hơn một lần thì đó không phải là LL(1), nếu trái lại thì văn phạm là LL(1).

- Cách khác: một văn phạm là LL(1) phải thoả mãn điều kiện sau:

Nếu $A \to \alpha \mid \beta$ là hai sản xuất của văn phạm đó thì phải thoả mãn:

a) $First(\alpha) \cap First(\beta) = \varnothing$

b) Nếu $\varepsilon \in First(\alpha)$ thì $Follow(A) \cap First(\beta) = \varnothing$

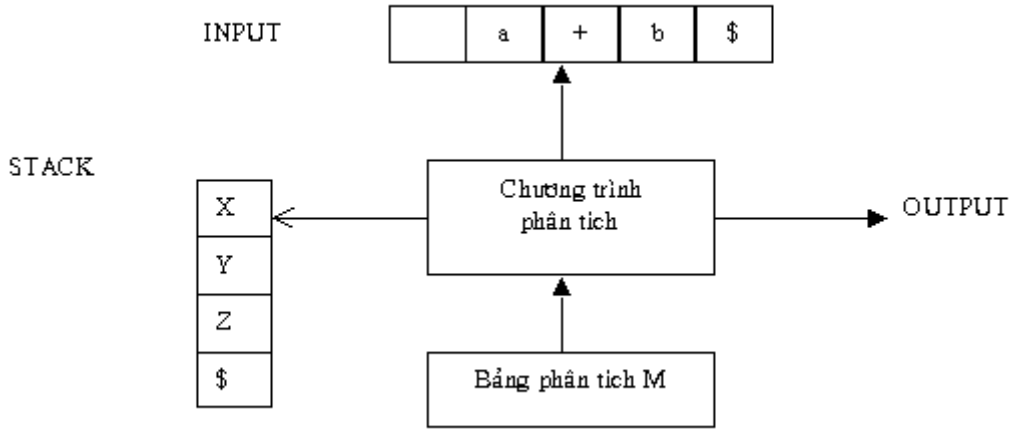
2.3.4 Thuật toán phân tích LL(1)

* *Mô tả*: Cơ sở của phân tích LL là dựa trên phương pháp phân tích topdown và máy ôtômát đẩy xuống.

- Vùng đệm chứa xâu vào với cuối xâu là ký hiệu kết thúc xâu $.

- Ngăn xếp chứa các ký hiệu văn phạm thể hiện quá trình phân tích. Đáy ngăn xếp kí hiệu $.

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



-Mô hình của phân tích cú pháp LL

Tại thời điểm hiện tại, giả sử X là ký hiệu trên đỉnh ngăn xếp và a là ký hiệu đầu vào. Các hành động điều khiển được thực hiện như sau:

1. Nếu X = a = $, quá trình phân tích thành công
2. Nếu X = a ≠ $, lấy X ra khỏi ngăn xếp và dịch con trỏ đầu vào đến ký hiệu tiếp theo
3. Nếu X là một ký hiệu không kết thúc, xét mục M(X,a) trong bảng phân tích. Có hai trường hợp xảy ra:

a) Nếu M(A,a) = X -> $Y_1 . . . Y_k$ thì lấy X ra khỏi ngăn xếp và đẩy vào ngăn xếp $Y_1, . . ., Y_k$ theo thứ tự ngược lại (để ký hiệu được phân tích tiếp theo trên đỉnh ngăn xếp phải là Y_1, tạo ra dẫn xuất trái).

b) Nếu M(A,a) là lỗi: thì phân tích gặp lỗi và gọi bộ khôi phục lỗi.

Ví dụ: Cho văn phạm:

{E→TE'; E'→ +TE' | ε; T→FT'; T'→*FT' | ε; F→ (E) | id}

a) Tính First và Follow cho các ký hiệu không kết thúc.
b) Tính First cho vế phải của các sản xuất.
c) Xây dựng bảng phân tích LL(1) cho văn phạm trên
d) Phân tích LL đối với xâu vào "id+id*id"

Ký hiệu văn phạm	First	Follow
E	*(, id*	*), $*
E'	*+, ε*	*), $*
T	*(, id*	*+,), $*
T'	**, ε*	*+,), $*
F	*(, id*	*+, *,), $*

Sản xuất	First của vế phải
E→TE’	*(, id*
E’→+TE’	+
T→FT’	*(, id*
T’→*FT’	*
F→(E)	*(*
F→id	*Id*

Bảng phân tích LL(1)

Ký hiệu Vế trái	Ký hiệu đầu vào					
	Id	+	*	(	)	$
E	E→TE’			E → TE’		
E’		E’→ +TE’			E’→ ε	E’→ ε
T	T→FT’			T→FT’		
T’		T’→ ε	T’→*FT’		T’→ ε	T’→ ε
F	F→ id			F→ (E)		

Phân tích LL(1) cho xâu vào **“id+id*id”**

Ngăn xếp	Xâu vào	Đầu ra
$E	id+id*id$	E→TE’
$E’T	id+id*id$	T→FT’
$E’T’F	id+id*id$	F→id
$E’T’id	id+id*id$	rút gọn id
$E’T’	+id*id$	T’→ε
$E’	+id*id$	E’→+TE’
$E’T+	+id*id$	rút gọn +
$E’T	id*id$	T→FT’
$E’T’F	id*id$	F→id
$E’T’id	id*id$	rút gọn id
$E’T’	*id$	T’→*FT’
$E’T’F*	*id$	rút gọn *
$E’T’F	id$	F→id

$E'T'id	id$	rút gọn id
$E'T'	$	T'→ε
$E'	$	E'→ε
$	$	

Từ bảng phân tích, chúng ta có suy dẫn trái như sau:

E=>TE'=>FT'E'=>idT'E'=>idE'=>id+TE'=>id+FT'E'=>id+idT'E'=>id+id *FT'E'=> id+id*idT'E'=>id+id*idE'=>id=id*id .

2.4 Phân tích LR(k)

- L là Left to right: quét xâu vào từ trái quá phải.
- R là Right most parsing: suy dẫn sinh ra là suy dẫn phải.
- k là số ký hiệu nhìn trước để đưa ra quyết định phân tích.

** Ưu điểm:*

- Nhận biết được tất cả các cấu trúc của ngôn ngữ lập trình được tạo ra dựa theo các văn phạm phi ngữ cảnh.
- LR là phương pháp phân tích cú pháp gạt - thu gọn không quay lui tổng quát nhất đã được biết đến nhưng lại có thể được cài đặt hiệu quả như những phương pháp gạt - thu gọn khác.
- Lớp văn phạm phân tích được nhờ phương pháp LR là một tập bao hàm thực sự của lớp văn phạm phân tích được bằng cách phân tích cú pháp dự đoán.
- Phát hiện được lỗi cú pháp ngay khi có thể trong quá trình quét đầu vào từ trái sang.

** Nhược điểm:*

Ta phải thực hiện quá nhiều công việc để xây dựng được bộ phân tích LR cho một ngôn ngữ lập trình.

2.4.2 Một số khái niệm

1) <u>Mục</u> (Item)

Cho một văn phạm G. Với mỗi sản xuất *A->xy*, ta chèn dấu chấm vào tạo thành *A->x .y* và gọi kết quả này là một mục.

Mục *A->x.y* cho biết qúa trình suy dẫn sử dụng sản xuất *A->xy* và đã suy dẫn đến hết phần x trong sản xuất, quá trình suy dẫn tiếp theo đối với phần xâu vào còn lại sẽ bắt đầu từ y.

Ví dụ: Luật sinh A → XYZ có 4 mục như sau:

A→ •XYZ A→ X•YZ A→ XY•Z A→ XYZ•

Luật sinh A → ε chỉ tạo ra một mục A → •

2) <u>Mục có nghĩa</u> (valid item)

Một mục $A \rightarrow \beta_1.\beta_2$ gọi là mục có nghĩa(valid item) đối với tiền tố khả tồn $\alpha\beta_1$ nếu tồn tại một dẫn xuất: $S =>^*_{rm} \alpha Aw =>_{rm} \alpha\beta_1\beta_2 w$

Tập tất cả các mục có nghĩa đối với tiền tố khả tồn gọi là tập I.

Một tập mục có nghĩa đối với một tiền tố khả tồn nói lên rất nhiều điều trong quá trình suy dẫn gạt - thu gọn: Giả sử quá trình gạt thu gọn đang ở trạng thái với ngăn xếp là x và phần ký hiệu đầu vào là v (*)

ngăn xếp	đầu vào
$x	v$

Thế thì, quá trình phân tích tiếp theo sẽ phụ thuộc vào tập mục có nghĩa I của tiền tố khả tồn thuộc x. Với một mục $[A->\beta_1.\beta_2] \in I$, cho chúng ta biết x có dạng $\alpha\beta_1$, và quá trình phân tích phần còn lại w của xâu đầu vào nếu theo sản xuất $A->\beta_1\beta_2$ sẽ được tiếp tục từ β_2 của mục đó. Hành động gạt hay thu gọn sẽ phụ thuộc vào β_2 là rỗng hay không. Nếu

$\beta_2 = \varepsilon$ thì phải thu gọn β_1 thành A, còn nếu $\beta_2 \neq \varepsilon$ thì việc phân tích theo sản xuất

$A \rightarrow \beta_1\beta_2$ đòi hỏi phải sử dụng hành động gạt.

- Mọi quá trình phân tích tiếp theo của trạng thái (*) đều phụ thuộc vào các mục có nghĩa trong tập các mục có nghĩa I của tiền tố khả tồn x.

- Có thể có nhiều mục có nghĩa đối với một tiền tố x. Các mục này có thể có các hành động xung đột (bao gồm cả gạt và thu gọn), trong trường hợp này bộ phân tích sẽ phải dùng các thông tin dự đoán, dựa vào việc nhìn ký hiệu đầu vào tiếp theo để quyết định nên sử dụng mục có nghĩa nào với tiền tố x (tức là sẽ cho tương ứng gạt hay thu gọn). Nếu quá trình này cho những quyết định không xung đột (duy nhất) tại mọi trạng thái thì ta nói văn phạm đó phân tích được bởi thuật toán LR

- Tư tưởng của phương pháp phân tích LR là phải xây dựng được tập tất cả các mục có nghĩa đối với tất cả các tiền tố khả tồn.

3) <u>Tập chuẩn tắc LR(0)</u>

LR(0) là tập các mục có nghĩa cho tất cả các tiền tố khả tồn.

LR(0) theo nghĩa: LR nói lên đây là phương pháp phân tích LR, còn số 0 có nghĩa là số ký tự nhìn trước là 0.

4) <u>Văn phạm gia tố</u>(Augmented Grammar)(mở rộng)

Văn phạm G - ký hiệu bắt đầu S, thêm một ký hiệu bắt đầu mới S' và luật sinh S' → S để được văn phạm mới G' gọi là văn phạm gia tố.

Ta xây dựng văn phạm gia tố của một văn phạm theo nghĩa, đối với văn phạm ban đầu, quá trình phân tích sẽ bắt đầu bởi các sản xuất có vế trái là S. Khi đó, chúng ta xây dựng văn phạm gia tố G' thì mọi quá trình phân tích sẽ bắt đầu từ sản xuất S'→ S

5) Phép toán closure

Nếu I là một tập các mục của một văn phạm G thì closure(I) là tập các mục được xây dựng từ I bằng hai qui tắc sau:

1. Khởi đầu mỗi mục trong I đều được đưa vào closure(I)

2. Nếu [A -> α.Bβ] ∈ closure(I) và B->γ là một sản xuất thì thêm [B -> .γ] vào closure(I) nếu nó chưa có ở đó. Áp dụng qui tắc này đến khi không thêm được một mục nào vào closure(I) nữa.

Trực quan:

Nếu *[A -> α.Bβ]* là mục có nghĩa đối với tiền tố khả tồn $x\alpha$ và có $B \rightarrow \gamma$ là sản xuất ta cũng có *[B->.γ]* là mục có nghĩa đối với tiền tố khả tồn $x\alpha$.

Phép toán tính bao đóng của một mục là để tìm tất cả các mục có nghĩa tương đương của các mục trong tập đó.

Theo định nghĩa của một mục là có nghĩa đối với một tiền tố khả tồn, chúng ta có suy dẫn S =>*rm xAy =>rm xαBβy

Sử dụng sản xuất B -> γ ta có suy dẫn S =>*rm xαγβy. Vậy thì [B->.γ] là một mục có nghĩa của tiền tố khả tồn xα

Ví dụ: Xét văn phạm mở rộng của biểu thức:

E' → E E → E + T | T T → T * F | F F → (E) | id

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

E' → • E	(Luật 1)	E → • E + T	(Luật 2)
E → • T	(Luật 2)	T → • T * F	(Luật 2)
T → • F	(Luật 2)	F → • (E)	(Luật 2)
F → • id	(Luật 2)		

E' → • E được đặt vào closure(I) theo luật 1.

Khi có một E đi ngay sau một • , bằng luật 2 ta thêm các sản xuất E với các chấm nằm ở đầu trái (E → • E + T và E → • T).

Bây giờ lại có T đi theo sau một •, ta lại thêm T → • T * F và T → • F vào.

Cuối cùng ta có Closure(I) như trên.

6) Phép toán goto

goto(I,X) với I là một tập các mục và X là một ký hiệu văn phạm.

goto(I,X) được định nghĩa là bao đóng của tập tất cả các mục *[A->αX.β]* sao cho *[A->α.Xβ]* $\in I$.

Trực quan:

Nếu I là tập các mục có nghĩa đối với một tiền tố khả tồn γ nào đó thì goto(I,X) là tập các mục có nghĩa đối với tiền tố khả tồn *γX*.

gọi tập J=goto(I,X) thì cách tính tập J như sau:

1. Nếu [A->α.Xβ] ∈ I thì thêm [A->αX.β] vào J
2. J=closure(J)

Phép toán goto là phép phát triển để xây dựng tất cả các tập mục cho tất cả các tiền tố khả tồn có thể.

Ví dụ : Giả sử I = {E' → E•, E → E • + T} Tính goto (I, +) ?

Ta có J = { E→ E + • T} ⇒ goto (I, +) = closure(I') bao gồm các mục :

E → E + • T (Luật 1)	T → • F (Luật 2)
T → • T * F (Luật 2)	F → • (E) (Luật 2)
	F → • id (Luật 2)

Tính Goto(I,+) bằng cách kiểm tra I cho các mục với dấu + ở ngay bên phải chấm. E'→ E• không phải mục như vậy nhưng E → E • + T thì đúng. Ta chuyển dấu chấm qua bên kia dấu + để nhận được E → E + • T và sau đó tính closure cho tập này.

7) Thuật toán xây dựng LR(0)

Cho văn phạm G, văn phạm gia tố của nó là G'

Tập C là tập các tập mục LR(0) được tính theo thuật toán như sau:

1). C = {closure({[S'->.S]})}

2). Đối với mỗi mục I trong C và mỗi ký hiệu văn phạm X, tính goto(I,X). Thêm goto(I,X) vào C nếu không rỗng và không trùng với bất kỳ tập nào có trong C

Thực hiện bước 2 đến khi nào không thêm được tập nào nữa vào C

Ví dụ: Cho văn phạm G:

{ E → E + T | T; T → T * F | F; F → (E) | a }

Hãy tính LR(0)

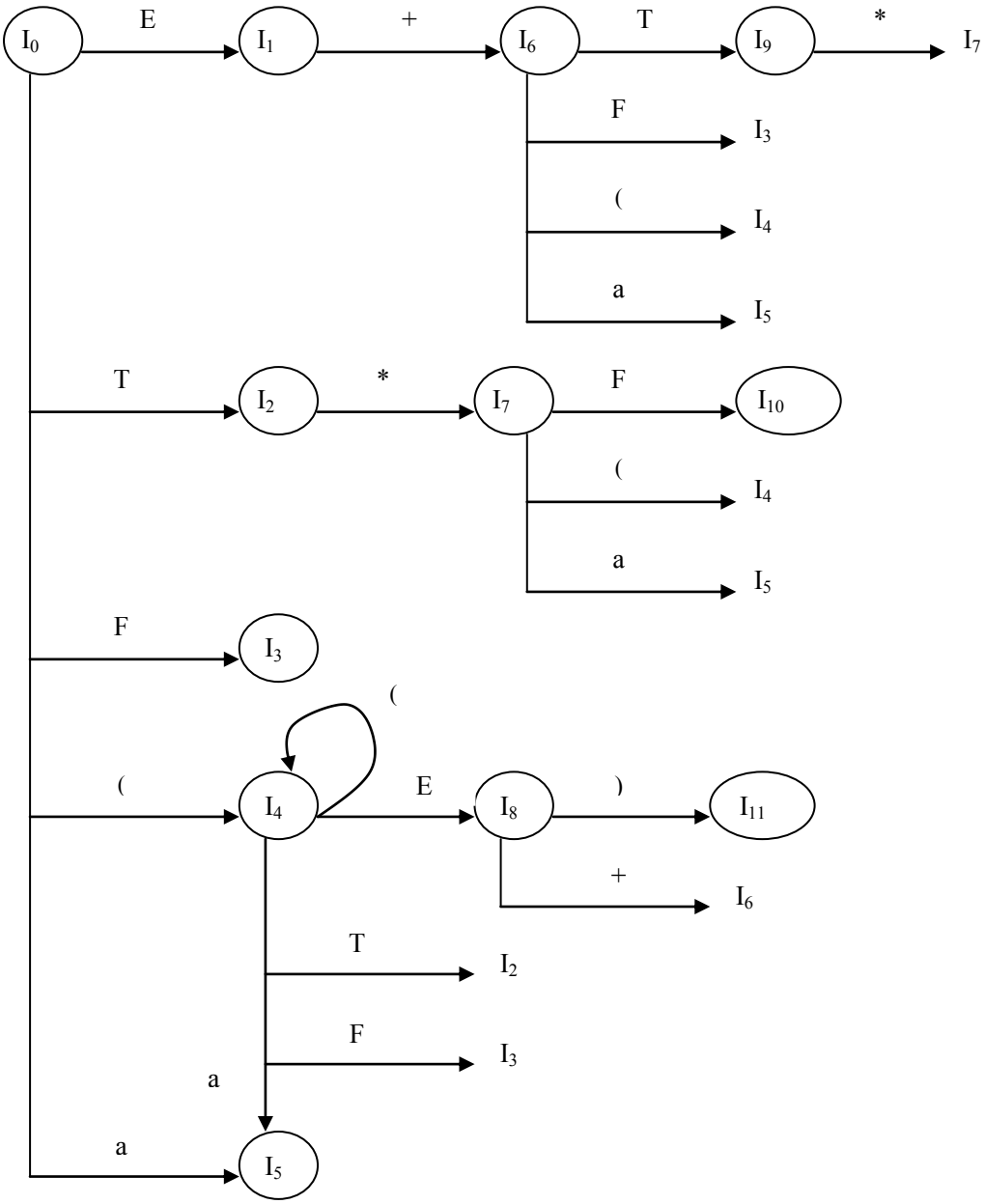
- Xét văn phạm G' là văn phạm gia tố của G. Văn phạm G:

{E' → E; E → E + T | T; T → T * F | F; F → (E) | a}

Tính theo thuật toán trên ta có kết quả như sau:

Closure({E'→ E}): **I_0: =** {E' → •E E → •E + T E → •T T → •T * F T → •F F → •(E) F → •a }	Goto (I_1, **+**) **I_6** = { E → E+•T T → •T*F T → •F F → •(E) F → •a}
Goto (I_0, E) **I_1=** { E' → E• E → E•+T}	Goto (I_2, ***) **I_7=** { T -> T*.F F -> .(E) F -> .a}
Goto (I_0, T) **I_2** = {E → T• T → T•*F}	Goto (I_4, E) **I_8** = {F → (E•) E → E• + T}
Goto (I_0, F) **I_3=** { T → F•}	Goto (I_4, T) **I_9**= { E → T• T → T•* F }
Goto (I_0, ( ) **I_4 =** {F → (•E) E → •E + T E → •T T → •T * F T →•F F → •(E) F → •a }	Goto (I_4, F) **I_{10}**= { T → F•}
Goto (I_0, id) **I_5= {** F → a•}	Goto (I_4, a) **I_{11}**= {E → E+T•}

Xây dựng tập C dựa trên hàm goto có thể được xem như một sơ đồ chuyển trạng thái của một DFA. Trong đó, I_0 là trạng thái xuất phát, bằng cách xây dựng các trạng thái tiếp bằng chuyển trạng thái theo đầu vào là các ký hiệu văn phạm. Đường đi của các ký hiệu đầu vào chính là các tiền tố khả tồn. Các trạng thái chính là tập các mục có nghĩa của các tiền tố khả tồn đó.



2.4.3 Xây dựng bảng phân tích SLR

Để xây dựng bảng phân tích LR ta có 3 phương pháp khác nhau:

+ Phương pháp Simple LR - SLR: là phương pháp "yếu" nhất nhưng dễ cài đặt nhất. Ta gọi bảng phân tích cú pháp tạo ra bởi phương pháp này là bảng SLR và bộ phân tích cú pháp tương ứng là bộ phân tích cú pháp SLR, với văn phạm tương đương là văn phạm SLR.

+ Phương pháp LR chuẩn (canonical LR) Phương pháp mạnh nhất nhưng tốn kém nhất.

+ Phương pháp LR nhìn vượt (LALR – LookaheadLR) là phương pháp trung gian về sức mạnh và chi phí giữ 2 phương pháp trên. Phương pháp này làm việc với hầu hết các văn phạm.

*** Xây dựng bảng phân tích SLR**

- Bảng phân tích bao gồm 2 phần : hàm **action** và hàm **goto.**

- ✓ action[s_m, a_i] có thể có một trong 4 giá trị :
  1. **shift s** : đẩy s, trong đó s là một trạng thái.
  2. **reduce (A→ β)** :thu gọn bằng luật sinh A→ β.
  3. **accept** : Chấp nhận
  4. **error** : Báo lỗi
- ✓ Goto lấy 2 tham số là một trạng thái và một ký hiệu văn phạm, nó sinh ra một trạng thái.

Cho văn phạm G, ta tìm văn phạm gia tố của G là G', từ G' xây dựng C là tập chuẩn các tập mục cho G', xây dựng hàm phân tích action và hàm nhẩy goto từ C bằng thuật toán sau.

Input: Một văn phạm tăng cường G'

Output: Bảng phân tích SLR với hàm action và goto

Phương pháp:

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

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

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

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

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

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

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

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

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

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

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

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

Dựa vào họ tập hợp mục C đã được xây dựng ở trên, ta thấy:

Xét tập mục **I_0** : Mục F → • (E) cho ra action[0, (] = "shift 4", và mục

F→ • **id** cho action[0, id] = "shift 5". Các mục khác trong $\mathbf{I_0}$ không sinh được hành động nào.

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

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

Tiếp tục theo cách này, ta thu được bảng phân tích cú pháp SLR(0):
Ký hiệu: si là shift i, rj là reduce theo luật (j), khoảng trống biểu thị lỗi

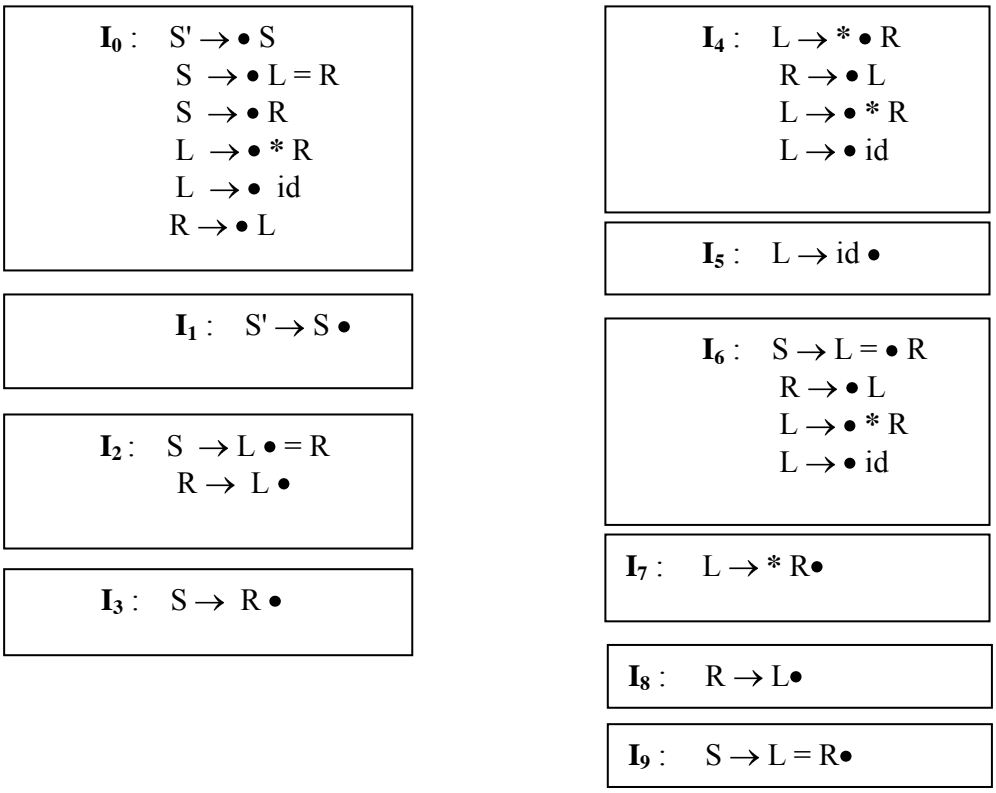
Trạng thái	**Action**						**goto**		
	a	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			S4					
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			S4			8	2	3
5		r6	r6		r6	r6			
6	s5			S4				9	3
7	s5			S4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

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

{S → L = R S → R L → * R L → **id** R → L}

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

Họ tập hợp các mục C bao gồm:



Khi xây dựng bảng phân tích SLR cho văn phạm, khi xét tập mục I2 ta thấy mục đầu tiên trong tập này làm cho action[2, =] = "shift 6". Bởi vì = $\in$ FOLLOW(R), nên mục thứ hai sẽ đặt action[2, =] = "reduce (R $\rightarrow$ L)" $\Rightarrow$ Có sự đụng độ tại action[2, =]. Vậy văn phạm trên không là văn phạm SLR(1).

2.4.4 Thuật toán phân tích LR

Phân tích LR là một thể phân tích cú pháp gạt - thu gọn, nhưng điểm khác biệt so với phân tích Bottom-up là nó không quay lui. Tại mỗi thời điểm nó xác định được duy nhất hành động gạt hay thu gọn.

*** Mô hình:** gồm các thành phần sau:

- Stack lưu một chuỗi $s_0X_1s_1X_2s_2 \ldots X_ms_m$ trong đó s_m nằm trên đỉnh Stack. X_i là một ký hiệu văn phạm, si là một trạng thái tóm tắt thông tin chứa trong Stack bên dưới nó.

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

Giải thuật

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

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

Phương pháp:

Khởi tạo s_0 là trạng thái khởi tạo nằm trong Stack và w$ nằm trong bộ đệm nhập.

Đặt ip vào ký hiệu đầu tiên của w$;

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

Ví dụ: Cho văn phạm:

(1) E -> E + T (2) E -> T (3) T -> T * F

(4) T -> F (5) F -> (E) (6) F -> a

Chúng ta sử dụng thuật toán LR để phân tích xâu vào "a*a+a". Áp dụng bảng phân tích trong phần 2.4.3, ta có quá trình phân tích như sau:

Ngăn xếp	**Đầu vào**	**Hành động**
0	id * id + id $	gạt
0 id 5	* id + id $	thu gọn F->id
0 F 3	* id + id $	thu gọn T->F
0 T 2	* id + id $	gạt
0 T 2 * 7	id + id $	gạt
0 T 2 * 7 id 5	+ id $	thu gọn F->id
0 T 2 * 7 F 10	+ id $	thu gọn T->T*F
0 T 2	+ id $	thu gọn E->T
0 E 1	+ id $	gạt

0 E 1 + 6	id $	gạt
0 E 1 + 6 id 5	$	thu gọn F->id
0 E 1 + 6 F 3	$	thu gọn T->F
0 E 1 + 6 T 9	$	thu gọn E->E+T
0 E 1	$	chấp nhận (accepted)

Quá trình phân tích LR

2.4.5 Xây dựng bảng phân tích LR chuẩn

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

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

Input : Văn phạm tăng cường G'

Output: Họ tập hợp các mục LR(1).

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

Function Closure (I);
begin
  Repeat
    For Mỗi mục [A → α • Bβ,a] trong I, mỗi luật sinh B → γ trong G và mỗi ký hiệu kết thúc b ∈ FIRST (βa) sao cho [B → • γ, b] ∉ I ***do***
      Thêm [B → • γ, b] vào I;
    Until Không còn mục nào có thể thêm cho I được nữa;
    return I;
end;
Function goto (I, X);
begin
    Gọi J là tập hợp các mục [A → αX•β, a] sao cho [A → α•Xβ, a]∈ I;
    return Closure(J);
end;
Procedure Items (G');
    C := Closure ({[S' → •S, $]})
    Repeat
    For Mỗi tập các mục I trong C và mỗi ký hiệu văn phạm X
    sao cho goto(I, X) ≠ ∅ và goto(I, X) ∉ C ***do***
      Thêm goto(I, X) vào C;
    Until Không còn tập các mục nào có thể thêm cho C;

Ví dụ: Xây dựng bảng LR chính tắc cho văn phạm gia tố G' có các luật sinh sau:

{ S' → S (1) S → L = R3 (2) S → R

(3) L → * R (4) L → id (5) R → L }

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

I_0 : S' → • S, $ Closure (S' → •S, $) {S → • L = R, $ S → • R, $ L → • * R, = L → • id, = R → • L, $}	Goto (I_4,R) **I_7** : L → * R•, = Goto (I_4, L) **I_8** : R→ L•, = Goto (I_6,R) **I_9** : S → L = R•, $ Goto (I_6,L) **I_{10}** :R → L•, $ Goto (I_6,*) **I_{11}** :L → * • R, $ R → • L, $ L → • * R, $ R → • id, $ Goto (I_6, id) **I_{12}** :L → id •, $ Goto (I_{11},R) **I_{13}** :L → * R•, $
Goto (I_0,S) **I_1** : {S' → S •, $ } Goto (I_0, L) **I_2** : {S → L • = R, $ R → L •, $} Goto ($I_0$,R) **$I_3$**: S → R •, $ Goto (I_0,*) **I_4**: L → * • R, = R → • L, = L → • * R, = R → • id, = Goto (I_0,id) **I_5** : L → id •, = Goto (I_2,=) **I_6** : S → L = • R, $ R → • L, $ L → • * R, $ L → • id, $	Goto (I_{11},L) ≡ **I_{10}** Goto (I_{11},*) ≡ **I_{11}** Goto (I_{11},id) ≡ I_{12}

2.4.6 Xây dựng bảng phân tích cú pháp LR chính tắc

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

Input: Văn phạm tăng cường G'

Output: Bảng LR với các hàm action và goto

Phương pháp:

1. Xây dựng C = { I0, I1, ... In } là họ tập hợp mục LR(1)

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

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

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

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

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

3. Nếu goto(Ii, A) = Ij thì goto[i,A] = j

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

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

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

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

Trạng thái	**Action**				**Goto**		
	=	*****	**id**	**$**	**S**	**L**	**R**
0		s_4	s_5		1	2	3
1				acc			
2	s_6			r_5			
3				r_2			
4		s_4	s_5			8	7
5	r_4						
6		s_{11}	s_{12}			10	9
7	r_3						
8	r_5						
9				r_1			
10				r_5			
11		s_{11}	s_{12}			10	13
12				r_4			
13				r_3			

2.4.5 Xây dựng bảng phân tích LALR

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

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

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

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

I4 và I11 I5 và I12 I7 và I13 I8 và I10

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

Input: Văn phạm tăng cường G'
Output: Bảng phân tích LALR

Phương pháp:

1. Xây dựng họ tập hợp các mục LR(1) C = {I0, I1, ..., In }

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

3. Đặt C' = {I0, I1, ..., Im } là kết quả thu được từ C bằng cách hợp các tập hợp có cùng hạt nhân. Action tương ứng với trạng thái i được xây dựng từ Ji theo cách thức như giải thuật 4.9.

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

4. Bảng goto được xây dựng như sau

Giả sử J = I1 $\cup$ I2 $\cup$ ... $\cup$ Ik . Vì I1, I2, ... Ik có chung một hạt nhân nên goto (I1,X), goto (I2,X), ..., goto (Ik,X) cũng có chung hạt nhân. Đặt K bằng hợp tất cả các tập hợp có chung hạt nhân với goto (I1,X) (goto(J, X) = K.

Ví dụ : Với ví dụ trên, ta có họ tập hợp mục C' như sau

C' = {I_0, I_1, I_2, I_3, I_{411}, I_{512}, I_6, I_{713}, I_{810}, I_9 }

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

closure ($S' \rightarrow \bullet S, \$$) : $S \rightarrow \bullet L = R, \$$
$S \rightarrow \bullet R, \$$
$L \rightarrow \bullet * R, =$
$L \rightarrow \bullet id, =$
$R \rightarrow \bullet L, \$$

$\mathbf{I_1}$ = Goto (I_0,S) : $S' \rightarrow S \bullet, \$$

$\mathbf{I_2}$ = Goto (I_0, L) : $S \rightarrow L \bullet = R, \$$
$R \rightarrow L \bullet, \$$

$\mathbf{I_3}$ = Goto (I_0,R) : $S \rightarrow R \bullet$

$\mathbf{I_{411}}$ = Goto (I_0,*), Goto (I_6,*) :
$L \rightarrow * \bullet R, = | \$$
$R \rightarrow \bullet L, = | \$$
$L \rightarrow \bullet * R, = | \$$
$R \rightarrow \bullet id, = | \$$

$\mathbf{I_{512}}$ = Goto (I_0,id), Goto (I_6,id) :
$L \rightarrow id \bullet, = | \$$

$\mathbf{I_6}$ = Goto(I_2,=) :
$S \rightarrow L = \bullet R, \$$
$R \rightarrow \bullet L, \$$
$L \rightarrow \bullet * R, \$$
$L \rightarrow \bullet id, \$$

$\mathbf{I_{713}}$ = Goto(I_{411}, R) :
$L \rightarrow * R\bullet, = | \$$

$\mathbf{I_{810}}$ = Goto(I_{411}, L), Goto(I_6, L):
$R \rightarrow L\bullet, = | \$$

$\mathbf{I_9}$ = Goto(I_6, R) :
$S \rightarrow L = R\bullet, \$$

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

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

Bảng phân tích cú pháp LALR

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

3 BẮT LỖI

* Giai đoạn phân tích cú pháp phát hiện và khắc phục được khá nhiều lỗi. Ví dụ lỗi do các từ tố từ bộ phân tích từ vựng không theo thứ tự của luật văn phạm của ngôn ngữ.

* Bộ bắt lỗi trong phần phân tích cú pháp có mục đích:

+ Phát hiện, chỉ ra vị trí và mô tả chính xác rõ rang các lỗi.

+ Phục hồi quá trìh phân tích sau khi gặp lỗi đủ nhanh để có thể phát hiện ra các lỗi tiếp theo.

+ Không làm giảm đáng kể thời gian xử lý các chương trình viết đúng.

* Các chiến lược phục hồi lỗi.

- Có nhiều chiến lược mà bộ phân tích có thể dùng để phục hồi quá trình phân tích sau khi gặp một lỗi cú pháp. Không có chiến lược nào tổng quát và hoàn hảo, có một số phương pháp dùng rộng rãi.

+ *Phục hồi kiểu trừng phạt*: Phương pháp đơn giản nhất và được áp dụng trong đa số các bộ phân tích. Mỗi khi phát hiện lỗi bộ phân tích sẽ bỏ qua một hoặc một số kí hiệu vào mà không kiểm tra cho đến khi nó gặp một kí hiệu trong tập từ tố đồng bộ. Các từ tố đồng bộ thường được xác định trước (VD: end, ;)

Người thiết kế chương trình dịch phải tự chọn các từ tố đồng bộ.

Ưu điểm: Đơn giản, không sợ bj vòng lặp vô hạn, hiệu quả khi gặp câu lệnh có nhiều lỗi.

+ *Khôi phục cụm từ*: Mỗi khi phát hienj lỗi, bộ phân tích cố gắng phân tích phần còn lại của câu lệnh. Nó có thể thay thế phần đầu của phần còn lại xâu này bằng một xâu nào đó cho phép bộ phân tích làm việc tiếp. Những việc này do người thiết kế chương trình dịch nghĩ ra.

+ *Sản xuất lỗi*: Người thiết kế phải có hiểu biết về các lỗi hường gặp và gia cố văn phạm của ngôn ngữ này tại các luật sinh ra cấu trúc lỗi. Dùng văn phạm này để khôi phục bộ phân tích. Nếu bọ phân tích dùng một luật lỗi có thể chỉ ra các cấu trúc lỗi phát hiện ở đầu vào.

3.1 Khôi phục lỗi trong phân tích tất định LL

* Một lỗi được phát hiện trong phân tích LL khi:

- Ký hiệu kết thúc nằm trên đỉnh ngăn xếp không đối sánh được với ký hiệu đầu vào hiện tại.

- Mục M(A,a) trong bảng phân tích là lỗi (rỗng).

* Khắc phục lỗi theo kiểu trừng phạt là bỏ qua các ký hiệu trên xâu vào cho đến khi xuất hiện một ký hiệu thuộc tập ký hiệu đã xác định trước gọi là tập ký hiệu đồng bộ. Xét một số cách chọn tập đồng bộ như sau:

a) Đưa các ký hiệu Follow(A) vào tập đồng bộ hoá của ký hiệu không kết thúc A. Nếu gặp lỗi, bỏ qua các từ của xâu vào cho đến khi gặp một phần tử của Follow(A) thì lấy A ra khỏi ngăn xếp và tiếp tục phân tích.

b) Đưa các ký hiệu trong First(A) vào tập đồng bộ hoá của ký hiệu không kết thúc A. Nếu gặp lỗi, bỏ qua các từ tố của xâu vào cho đến khi gặp một phần tử thuộc First(A) thì quá trình phân tích được tiếp tục.

Ví dụ: Ta phân tích xâu vào có lỗi là "**)id*+id**" với tập đồng bộ hoá của các ký hiệu không kết thúc được xây dựng từ tập First và tập Follow của ký hiệu đó.

Ngăn xếp	Xâu vào	Hành động
$E	)id*+id$	M(E,)) = lỗi, bỏ qua ')' để gặp id ∈ First(E)
$E	id*+id$	E->TE'
$E'T	id*+id$	T->FT'
$E'T'F	id*+id$	F->id
$E'T'id	id*+id$	rút gọn id
$E'T'	*+id$	T'->*FT'
$E'T'F*	*+id$	rút gọn *
$E'T'F	+id$	M(F,+) = lỗi, bỏ qua. Tại đây xảy ra hai trường hợp(ta chọn a): a).bỏ qua + vì id ∈ First(F) b).bỏ qua F vì + ∈ Follow(F)
$E'T'F	id$	F->id
$E'T'id	id$	rút gọn id
$E'T'	$	T'->ε
$E'	$	E'->ε
$	$	

3.2 Khôi phục lỗi trong phân tích LR

Một bộ phân tích LR sẽ phát hiện ra lỗi khi nó gặp một mục báo lỗi trong bảng action (chú ý sẽ không bao giờ bộ phân tích gặp thông báo lỗi trong bảng goto). Chúng ta có thể thực hiện chiến lược khắc phục lỗi cố gắng cô lập đoạn câu chứa lỗi cú pháp: quét dọc xuống ngăn xếp cho đến khi tìm được một trạng thái s có một hành động goto trên một ký hiệu không kết thúc A ngay sau nó. Sau đó bỏ đi không hoặc nhiều ký hiệu đầu vào cho đến khi gặp một ký hiệu kết thúc a thuộc Follow(A), lúc này bộ phân tích sẽ đưa trạng thái goto(s,A) vào ngăn xếp và tiếp tục quá trình phân tích.

Đọc thêm

Phương pháp phân tích bảng CYK (Cocke – Younger – Kasami)

- Giải thuật làm việc với tất cả các VP PNC. Thời gian phân tích là: n^3 (n là độ dài xâu vào cần phân tích), nếu văn phạm không nhập nhằng thì thờI gian phân tích là: n^2.

- Điều kiện của thuật toán là văn phạm PNC ở dạng chuẩn chômsky (CNF) và không có ε sản xuất (sản xuất $A \to \varepsilon$) và các kí hiệu vô ích.

Giải thuật CYK:

- Tạo một hình tam giác (mỗi chiều có độ dài là n , n là độ dài của xâu). Thực hiện giải thuật:

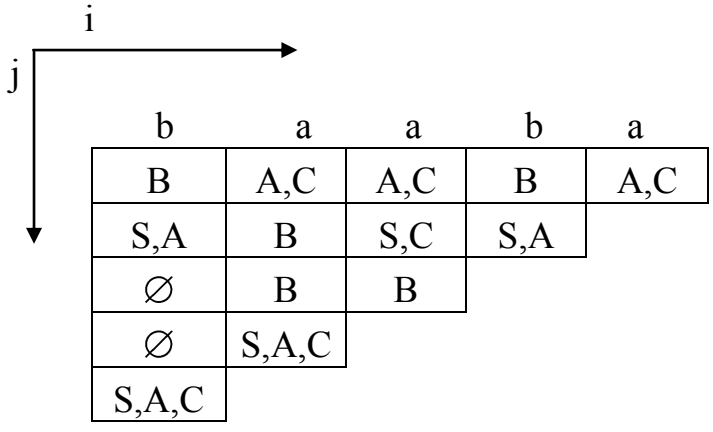
```
Begin
    1) For i:=1 to n do
    Δij = { A | A → a là một sản xuất và a là kí hiệu thứ i trong w};
    2) For j:=2 to n do
        For i:=1 to (n – j +1) do
          Begin
            Δij = ∅;
            For k:=1 to (j -1) do
            Δij = Δij ∪ { A | A → BC là một sản xuất; B ∈ Δik C∈ Δi+k, j -k };
          end;
 end;
```

Ví dụ: Xét văn phạm chuẩn chômsky

$S \to AB|BC$; $A \to BA|a$; $B \to CC|b$; $C \to AB|a$;

(1) (2) (3) (4) (5) (6) (7) (8)

Xâu vào w= baaba;



B	A,C	A,C	B	A,C
S,A	B	S,C	S,A	
∅	B	B		
∅	S,A,C			
S,A,C				

- Quá trình tính Δ_{ij} . VD: tính Δ_{24} , Tính:

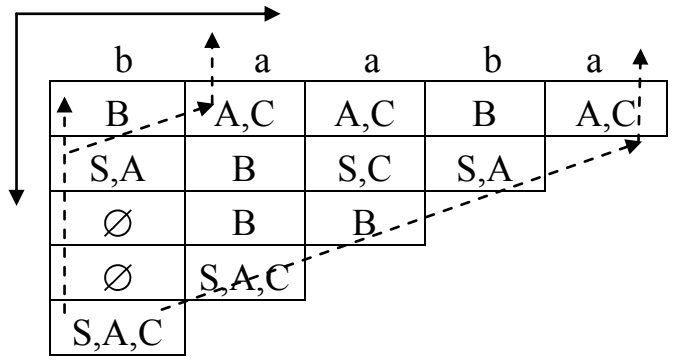
$\Delta_{21} = \{A,C\}$, $\Delta_{33} = \{B\}$, $\Delta_{21}\Delta_{33} = \{AB,CB\}$ Do (1), (7) nên đưa S,C vào Δ_{24}.

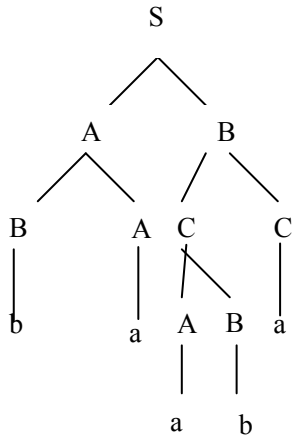
$\Delta_{22} = \{B\}$, $\Delta_{42} = \{S,A\}$, $\Delta_{22}\Delta_{42} = \{BS,BA\}$ Do (3) nên đưa A vào Δ_{24}.

$\Delta_{23} = \{B\}$, $\Delta_{51} = \{A,C\}$, $\Delta_{23}\Delta_{51} = \{BA,BC\}$ (2),(3) nên đưa S,C vào Δ_{24}.

Kết quả: $\Delta_{24} = \{S,A,C\}$.

- Nếu S ở ô cuối cùng thì ta kết luận: Xâu vào phân tích thành công và có thể dựng được cây phân tích cho nó. Số lượng cây phân tích = số lượng S có trong ô này.





BÀI TẬP

Câu 1

Tại sao cây phân tích nhận được từ thuật toán phân tích topdown được gọi là cây phân tích trái? Tại sao cây phân tích nhận được từ thuật toán phân tích bottom-up được gọi là cây phân tích phải?

Câu 2 Cho văn phạm G như sau:

{E → ! E E → E && E E → (E) E → a }

Với E là ký hiệu bắt đầu; ! , &&, a là các ký hiệu kết thúc.

1) Hỏi xâu vào "! (a) && a" có thuộc ngôn ngữ sinh ra bởi văn phạm G không? vì sao?

2) Chứng tỏ rằng văn phạm trên là nhập nhằng.

3) Văn phạm trên có phân tích được bởi phương pháp Topdown không? tại sao?

Câu 3 Cho văn phạm dùng để mô tả biểu thức số học như sau:

{ E -> T E' E' -> + T E' | ε T -> F T'
T' -> * F T' | ε F -> a | (E) }

Với E, E', T, T', F là các ký hiệu không kết thúc, E là ký hiệu bắt đầu.

1) Hãy viết một dẫn xuất phải nhất đối với xâu vào "a+a*a" và vẽ cây phân tích.

2) Có sử dụng phương pháp phân tích Bottom-up đối với văn phạm trên được không? Vì sao?

3) Hãy viết quá trình phân tích xâu vào "a+a" bằng phương pháp phân tích Top-down.

Câu 4 Cho văn phạm G như sau:

{ S -> A B A -> a A | ε B -> b B | ε }

Trong đó S, A, B là các ký hiệu không kết thúc, S là ký hiệu bắt đầu và a, b là các ký hiệu kết thúc.

1) Hãy tính First và Follow của S, A, B

2) Tính First của các vế phải của các sản xuất

3) Xây dựng bảng phân tích LL(1) cho văn phạm G. Văn phạm G có phải là văn phạm LL(1) không?

4) Nếu văn phạm trên là LL(1). Hãy viết quá trình phân tích LL(1) đối với xâu vào "aabb"

Câu 5 Chứng minh rằng văn phạm đệ qui trái không thể là LL(1).

Câu 6 Chứng minh rằng văn phạm LL(1) không thể nhập nhằng.

Câu 7 Cho văn phạm sau với N, B là các ký hiệu không kết thúc, N là ký hiệu bắt đầu:

(1) N → N B (2) N → B (3) B → a (4) B → b

Hãy viết quá trình phân tích LR với xâu vào "aaba" . Xây dựng bảng phân tích theo 3 phương pháp (SLR, LR(1), LALR)

Câu 8 Cho văn phạm sau với S, A, B là các ký hiệu không kết thúc, S là ký hiệu bắt đầu; a, b là các ký hiệu kết thúc.

{S -> AaAb | BbBa A -> ε B -> ε }

Xây dựng bảng phân tích LL(1). Văn phạm trên có phải là LL(1) không?

Văn phạm trên có phải là văn phạm SLR(1) không?

Câu 9 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)

Những văn phạm như thế nào thì không phân tích được bằng thuật toán bottom-up? Tại sao?

CHƯƠNG 4
BIÊN DỊCH DỰA CÚ PHÁP

Mục tiêu:

Sinh viên cần nắm được cách gắn các luật sinh của văn phạm (văn phạm biểu diễn cú pháp của ngôn ngữ) với các luật ngữ nghĩa (ý nghĩa của câu lệnh): định nghĩa cú pháp điều khiển và lược đồ dịch.

Biết cách thiết kế chương trình thực hiện một công việc nào đó từ một lược đồ dịch hay từ một định nghĩa cú pháp điều khiển cụ thể.

Nội dung:

- Định nghĩa cú pháp điều khiển

- Lược đồ dịch

1 MỤC ĐÍCH, NHIỆM VỤ

Quá trình dịch được bám theo cấu trúc cú pháp của chương trình nguồn cần dịch (cấu trúc cú pháp này được xác định thông qua bộ phân tích cú pháp).

Cú pháp-điều khiển (syntax-directed) là cơ chế điều khiển chương trình dịch bám theo cấu trúc cú pháp và được dùng cho các giai đoạn sau giai đoạn phân tích cú pháp (giai đoạn phân tích ngữ nghĩa, sinh mã trung gian …). Để thực hiện được cơ chế điều khiển này, ta sẽ gắn các hành động ngữ nghĩa của câu lệnh (luật ngữ nghĩa) vào các sản xuất của văn phạm được định nghĩa trong giai đoạn phân tích cú pháp (luật cú pháp). Sau đó, xây dựng cơ chế duyệt các hành động ngữ nghĩa này.

Có hai tiếp cận để liên kết các qui tắc ngữ nghĩa vào các luật cú pháp là Định nghĩa cú pháp điều khiển (syntax-directed definition) và lược đồ dịch (translation scheme).

- Định nghĩa cú pháp điều khiển:
  - là điều khiển ở mức cao
  - dấu đi nhiều chi tiết thực hiện
  - không cần chỉ rõ ràng thứ tự thực hiện.
- Lược đồ chuyển đổi
  - có thứ tự đánh giá luật ngữ nghĩa
  - cho phép thấy một số chi tiết thực hiện.

.

2 ĐỊNH NGHĨA CÚ PHÁP ĐIỀU KHIỂN

Cú pháp điều khiển (syntax-directed definition) là một dạng tổng quát hoá của văn phạm phi ngữ cảnh, trong đó mỗi ký hiệu văn phạm có một tập thuộc tính đi kèm.

Một cây phân tích cú pháp có trình bày các giá trị của các thuộc tính tại mỗi nút được gọi là cây phân tích cú pháp có chú giải (hay gọi là cây phân tích đánh dấu) (annotated parse tree).

Một thuộc tính là thứ bất kỳ: một xâu, một con số, một kiểu, một khoảng nhớ...Giá trị của một thuộc tính (tại một nút) được xác định bằng một luật ngữ nghĩa tương ứng với sản xuất dùng tại nút đó khi xây dựng cây phân tích cú pháp.

2.1 Định nghĩa cú pháp điều khiển

2.1.1 Dạng của định nghĩa cú pháp điều khiển

Trong mỗi cú pháp điều khiển, mỗi sản xuất $A \rightarrow \alpha$ có thể được liên kết với một tập các qui tắc ngữ nghĩa có dạng $b = f(c_1, \ldots, c_k)$ với f là một hàm và

a) b là một thuộc tính tổng hợp của A, còn $c_1, \ldots, c_k$ là các thuộc tính của các ký hiệu trong sản xuất đó. Hoặc

b) b là một thuộc tính kế thừa của một trong những ký hiệu ở vế phải của sản xuất, còn $c_1, \ldots, c_k$ là thuộc tính của các ký hiệu văn phạm.

Ta nói là thuộc tính b phụ thuộc vào các thuộc tính $c_1, \ldots, c_k$.

- Một văn phạm thuộc tính (Attribute Grammar) là một cú pháp điều khiển mà các luật ngữ nghĩa không có hành động phụ.

Ví dụ: Sau đây là văn phạm cho một chương trình máy tính bỏ túi với val là một thuộc tính biểu diễn giá trị của ký hiệu văn phạm.

Sản xuất	**Luật ngữ nghĩa**
L -> E **n**	*Print(E.val)*
E -> E_1 + T	*E.val = E_1.val + T.val*
E -> T	*E.val = T.val*
T -> T_1 * F	*T.val = T_1.val * F.val*
T -> F	*T.val = F.val*
F -> (E)	*F.val = E.val*
F -> **digit**	*F.val = digit.lexval*

Từ tố digit có thuộc tính Lexval: là giá trị của digit đó được tính nhờ bộ phân tích từ vựng. Kí hiệu n : xuống dòng, Print : in kết quả ra màn hình.

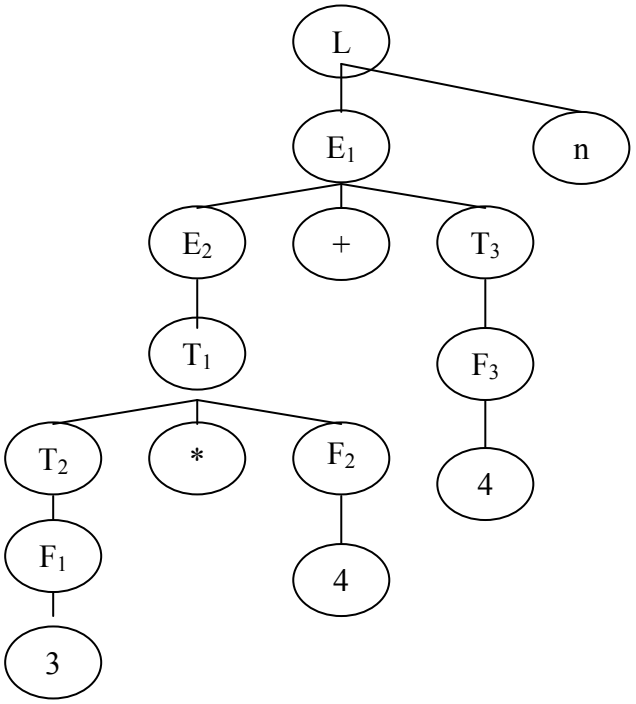
2.1.2 Thuộc tính tổng hợp

Trên một cây phân tích, thuộc tính tổng hợp được tính dựa vào các thuộc ở các nút con của nút đó hay nói cách khác thuộc tính tổng hợp được tính cho các ký hiệu ở vế trái của sản xuất và tính dựa vào thuộc tính của các ký hiệu ở vế phải.

Một cú pháp điều khiển chỉ sử dụng các thuộc tính tổng hợp được gọi là *cú pháp điều khiển thuần tính S* (S-attribute definition).

Một cây phân tích cho văn phạm cú pháp điều khiển thuần tính S có thể thực hiện các luật ngữ nghĩa theo hướng từ lá đến gốc và có thể sử dụng trong phương pháp phân tích LR.

Ví dụ: Vẽ cây cho đầu vào: 3*4+4n



Chúng ta duyệt và thực hiện các hành động ngữ nghĩa của ví dụ trên theo đệ qui trên xuống: khi gặp một nút ta sẽ thực hiện tính thuộc tính tổng hợp của các con của nó rồi thực hiện hành động ngữ nghĩa trên nút đó. Nói cách khác, khi phân tích cú pháp theo kiểu bottom-up, thì khi nào gặp hành động thu gọn, chúng ta sẽ thực hiện hành động ngữ nghĩa để đánh giá thuộc tính tổng hợp

F_1.val=3 (syntax: F_1->3 semantic: F_1.val=3.lexical)

F_2.val=4 (syntax: F_2->3 semantic: F_2.val=4.lexical)

T_2.val=3 (syntax: T_2->F_1 semantic: T_2.val=F_1.val)

T_1.val=3*4=12 (syntax: T_1->T_2*F_2 semantic: T_1.val=T_2.val*F_2.val)

F_3.val=4 (syntax: F_3->4 semantic: F_3.val=4.lexical)

T_3.val=4 (syntax: T_3->F_3 semantic: T_3.val=F_3.val)

E_1.val=12+4=16 (syntax: E_1->E_2+T_3 semantic: E_1.val=E_2.val+T_3.val)

"16" (syntax: L->E_1 n semantic: print(E_1.val))

2.1.3 Thuộc tính kế thừa

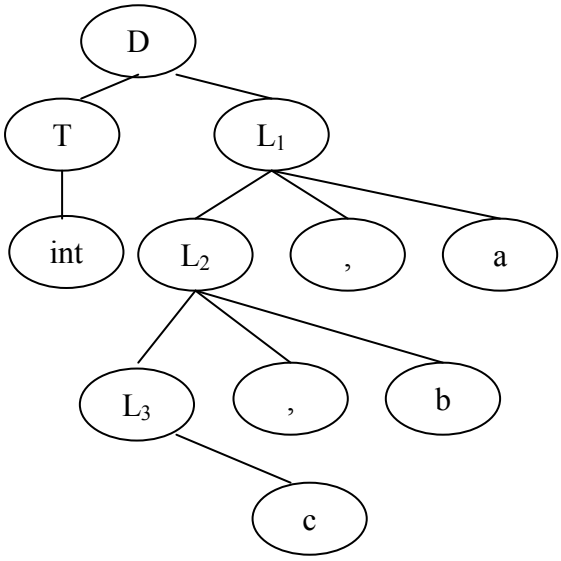
Thuộc tính kế thừa (inherited attribute) là thuộc tính tại một nút có giá trị được xác định theo giá trị thuộc tính của cha hoặc anh em của nó.

Thuộc tính kế thừa rất có ích trong diễn tả sự phụ thuộc ngữ cảnh. Ví dụ chúng ta có thể xem một định danh xuất hiện bên trái hay bên phải của toán tử gán để quyết định dùng địa chỉ hay giá trị của định danh.

Ví dụ về khai báo biến trong ngôn ngữ C

sản xuất	luật ngữ nghĩa
D -> T L	L.in := T.type
T -> int	T.type := interger
T -> float	T.type := float
L -> L_1, id	L_1.in := L.in ; addtype(id.entry, L.in)
L -> id	addtype(id.entry,L.in)

Ví dụ: int a,b,c Ta có cây cú pháp:



Chúng ta duyệt và thực hiện các hành động ngữ nghĩa sẽ được kết quả như sau:

T.type = interger (syntax:T->int semantic: T.type=interger)

L_1.in = interger (syntax: D -> T L_1 semantic: L_1.in=T.type)

L_2.in = interger (syntax: L_1 -> L_2 , a semantic: L_2.in = L_1.in)

a.entry = interger (syntax: L_1 -> L_2 , a semantic: addtype(a.entry,L_1.in))

L_3.in = interger (syntax: L_2 -> L_3 , b semantic: L_3.in = L_2.in)

b.entry = interger (syntax: L_2 -> L_3 , b semantic: addtype(b.entry,L_2.in))

c.entry = interger (syntax: L_3 -> c semantic: addtype(c.entry,L_3.in))

3 ĐỒ THỊ PHỤ THUỘC

Nếu một thuộc tính b tại một nút trong cây phân tích cú pháp phụ thuộc vào một thuộc tính c, thế thì hành động ngữ nghĩa cho b tại nút đó phải được thực hiện

sau khi thực hiện hành động ngữ nghĩa cho c. Sự phụ thuộc qua lại của các thuộc tính tổng hợp và kế thừa tại các nút trong một cây phân tích cú pháp có thể được mô tả bằng một đồ thị có hướng gọi là đồ thị phụ thuộc (dependency graph).

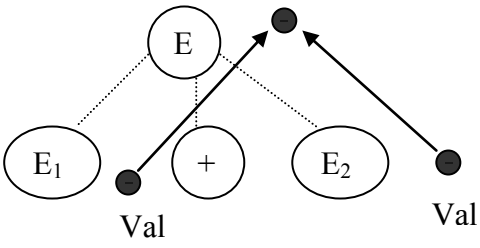
- Đồ thị phụ thuộc là một đồ thị có hướng mô tả sự phụ thuộc giữa các thuộc tính tại mỗi nút của cây phân tích cú pháp.

Trước khi xây dựng một đồ thị phụ thuộc cho một cây phân tích cú pháp, chúng ta chuyển mỗi hành động ngữ nghĩa thành dạng b := $f(c_1,c_2,\ldots,c_k)$ bằng cách dùng một thuộc tính tổng hợp giả b cho mỗi hành động ngữ nghĩa có chứa một lời gọi thủ tục. Đồ thị này có một nút cho mỗi thuộc tính, một cạnh đi vào một nút cho b từ một nút cho c nếu thuộc tính b phụ thuộc vào thuộc tính c. Chúng ta có thuật toán xây dựng đồ thị phụ thuộc cho một văn phạm cú pháp điều khiển như sau:

for mỗi nút n trong cây phân tích cú pháp **do**
    for mỗi thuộc tính a của ký hiệu văn phạm tại n **do**
        xây dựng một nút trong đồ thị phụ thuộc cho a;
for mỗi nút n trong cây phân tích cú pháp **do**
    for mỗi hành động ngữ nghĩa b:=$f(c_1,c_2,\ldots,c_k)$
        đi kèm với sản xuất được dùng tại n **do**
        for i:=1 to k **do**
            xây dựng một cạnh từ nút c_i đến nút b

VD 1: Dựa vào cây phân tích (nét đứt đoạn) và luật ngữ nghĩa ứng với sản xuất ở bảng, ta thêm các nút và cạnh thành đồ thị phụ thuộc:

Sản xuất	Luật ngữ nghĩa
$E \rightarrow E_1 \mid E_2$	$E.val = E_1.val + E_2.val$



Ví dụ 2:

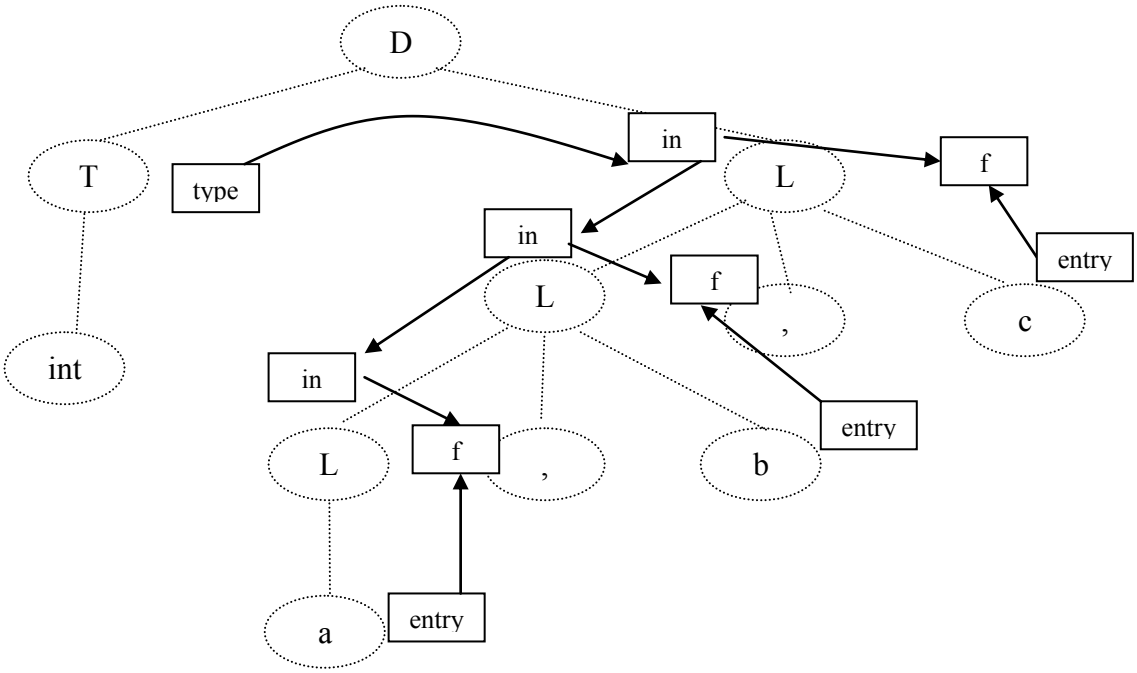
Với ví dụ 2, ta có một đồ thị phụ thuộc như sau:

chú ý:

+ chuyển hành động ngữ nghĩa *addentry(id.entry,L.in)* của sản xuất *L->L , id* thành thuộc tính giả *f* phụ thuộc vào *entry* và *in*

sản xuất	luật ngữ nghĩa

D -> T L	L.in := T.type
T -> int	T.type := interger
T -> float	T.type := float
L -> L_1, id	L_1.in := L.in ; addtype(id.entry, L.in)
L -> id	addtype(id.entry,L.in)



Đối với một đồ thị tổng quát, chúng ta phải để ý đến các đặc điểm sau:

+ Xây dựng đồ thị phụ thuộc cho các thuộc tính của ký hiệu văn phạm phải được xây dựng trên cây cú pháp.

+ Trong đồ thị phụ thuộc, mỗi nút đại diện cho một thuộc tính của một ký hiệu văn phạm.

+ Có thể một loại thuộc tính này lại phụ thuộc vào một loại thuộc tính khác, chứ không nhất thiết là chỉ các thuộc tính cùng loại mới phụ thuộc vào nhau. Trong ví dụ trên, thuộc tính *entry* phụ thuộc vào thuộc tính *in*.

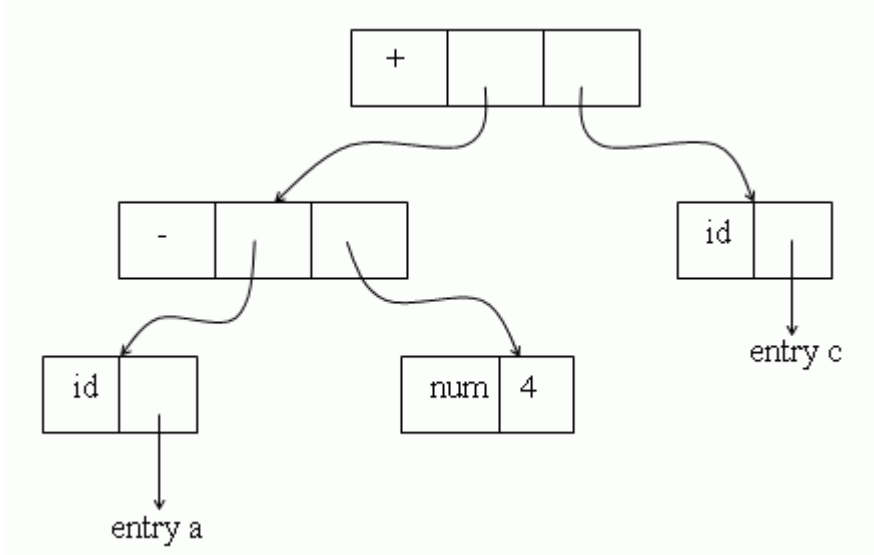
+ Có thể có "vòng" trong đồ thị phụ thuộc, khi đó chúng ta sẽ không tính được giá trị ngữ nghĩa cho các nút vì gặp một hiện tượng khi tính a cần tính b, mà khi tính b lại cần tính a.

Chính vì vậy, trong thực tế chúng ta chỉ xét đến văn phạm cú pháp ngữ nghĩa mà đồ thị phụ thuộc của nó là một DAG không có vòng.

4 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.

4.1. Xây dựng cây cú pháp cho biểu thức

Xây dựng cây con cho biểu thức con bằng cách tạo ra một nút cho toán hạng và toán tử. Con của nút toán tử là gốc của cây con biểu diễn cho biểu thức con toán hạng của toán tử đó.

Mỗi một nút có thể cài đặt bằng một mẩu tin có nhiều trường.

Trong nút toán tử, có một trường chỉ toán tử như là nhãn của nút, các trường còn lại chứa con 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. *mknode(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: Để xây dựng cây cú pháp cho biểu thức: a - 4 + c ta dùng một dãy các lời gọi các hàm nói trên.

(1): p1 := mkleaf(id, entrya)

(2): p2 := mkleaf(num,4)

(3): p3 := mknode(‘-‘, p1, p2)

(4): p4 := mkleaf(id, entryc)

(5): p5 := mknode(‘+’, p3, p4)

Cây được xây dựng từ dưới lên

entrya là con trỏ, trỏ tới ô của a trong bảng ký hiệu

entryc là con trỏ, trỏ tới ô của c trong bảng ký hiệu

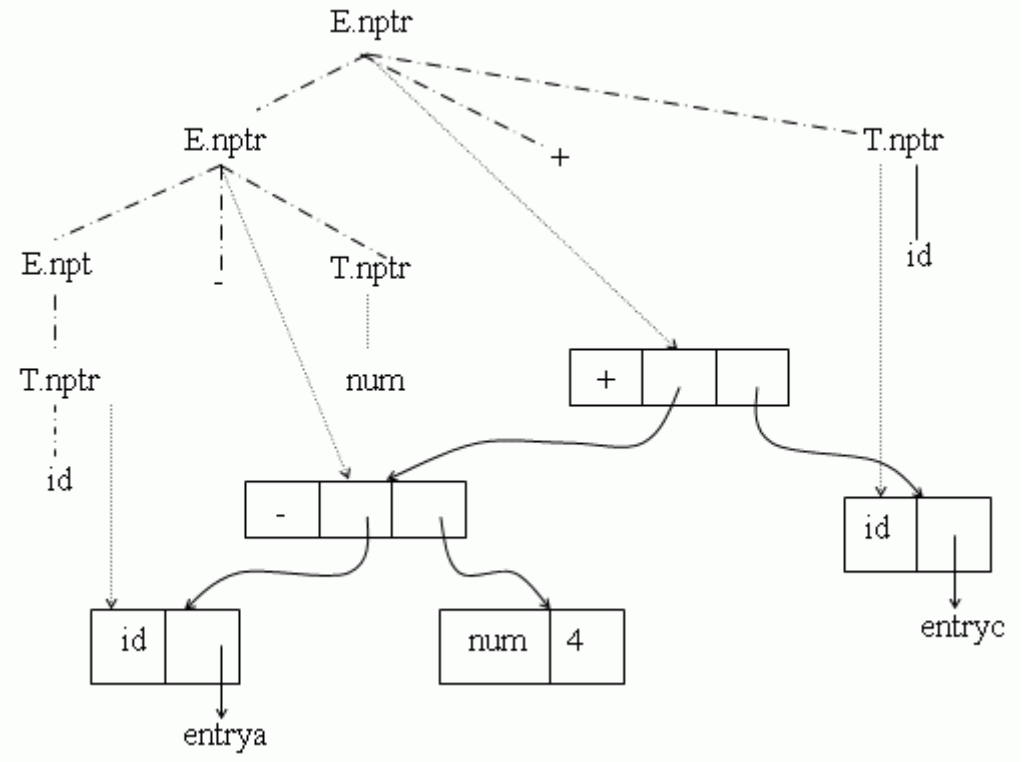
4.2 Xây dựng cây cú pháp từ định nghĩa trực tiếp cú pháp

Căn cứ vào các luật sinh văn phạm và luật ngữ nghĩa kết hợp mà ta phân bổ việc gọi các hàm mknode và mkleaf để tạo ra cây cú pháp.

Ví dụ: Định nghĩa trực tiếp cú pháp để xây dựng cây cú pháp cho biểu thức:

Luật sinh	Luật ngữ nghĩa
E → E_1 + T	*E.nptr := mknode('+', E_1.nptr, T.nptr)*
E → E_1 - T	*E.nptr := mknode('-', E_1.nptr, T.nptr)*
E → T	*E.nptr := T.nptr*
T → (E)	*T.nptr := E.nptr*
T → **id**	*T. nptr := mkleaf(id, id.entry)*
T → **num**	*T.nptr := mkleaf(num, num.val)*

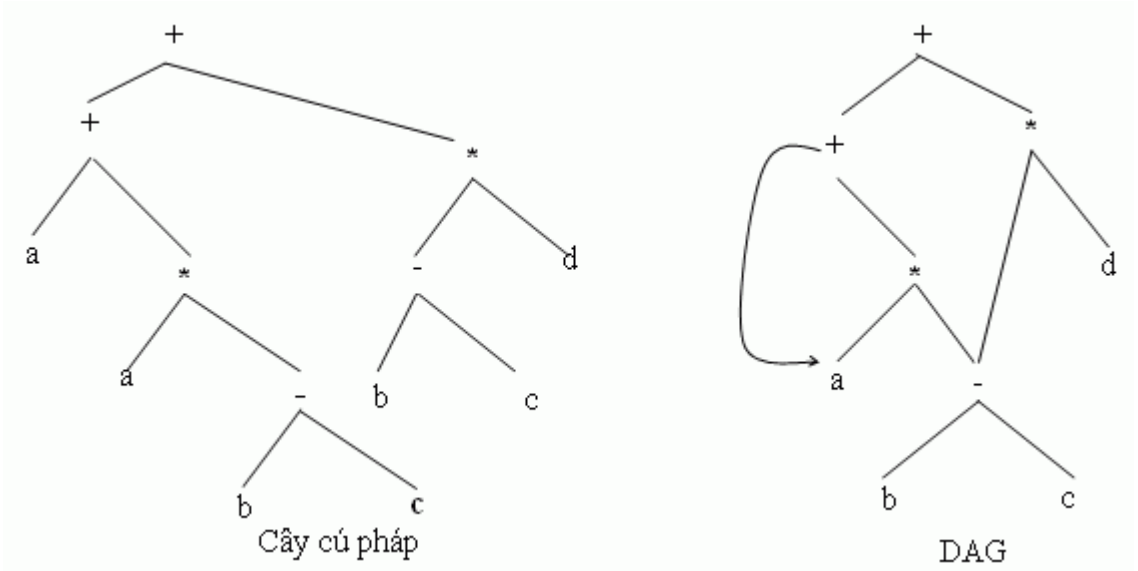
Luật ngữ nghĩa cho phép tạo ra cây cú pháp. Cây cú pháp có ý nghĩa về mặt cài đặt còn cây phân tích cú pháp chỉ có ý nghĩa về mặt logic.



5 ĐỒ THỊ CÓ HƯỚNG KHÔNG TUẦN HOÀN (Directed Acyclic Graph - DAG)

DAG cũng giống như cây cú pháp, tuy nhiên trong cây cú pháp các biểu thức con giống nhau được biểu diễn lặp lại còn trong DAG thì không. Trong DAG, một nút con có thể có nhiều "cha".

Ví dụ: Cho biểu thức a + a * (b - c) + (b - c) * d

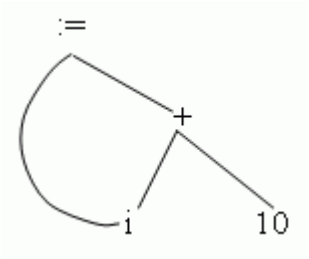


Để xây dựng một DAG, trước khi tạo một nút phải kiểm tra xem nút đó đã tồn tại chưa, nếu đã tồn tại thì hàm tạo nút (mknode, mkleaf) trả về con trỏ của nút đã tồn tại, nếu chưa thì tạo nút mới.

Cài đặt DAG: Người ta thường sử dụng một mảng mẩu tin , mỗi mẩu tin là một nút. Ta có thể tham khảo tới nút bằng chỉ số của mảng.

Lệnh gán DAG

i := i + 10



Biểu

id	entrya	
num	10	
+	1	2
:=	1	3

diễn

Nút 1: có nhãn là id, con trỏ trỏ tới entry i.

Nút 2: có nhãn là num, giá trị là 10.

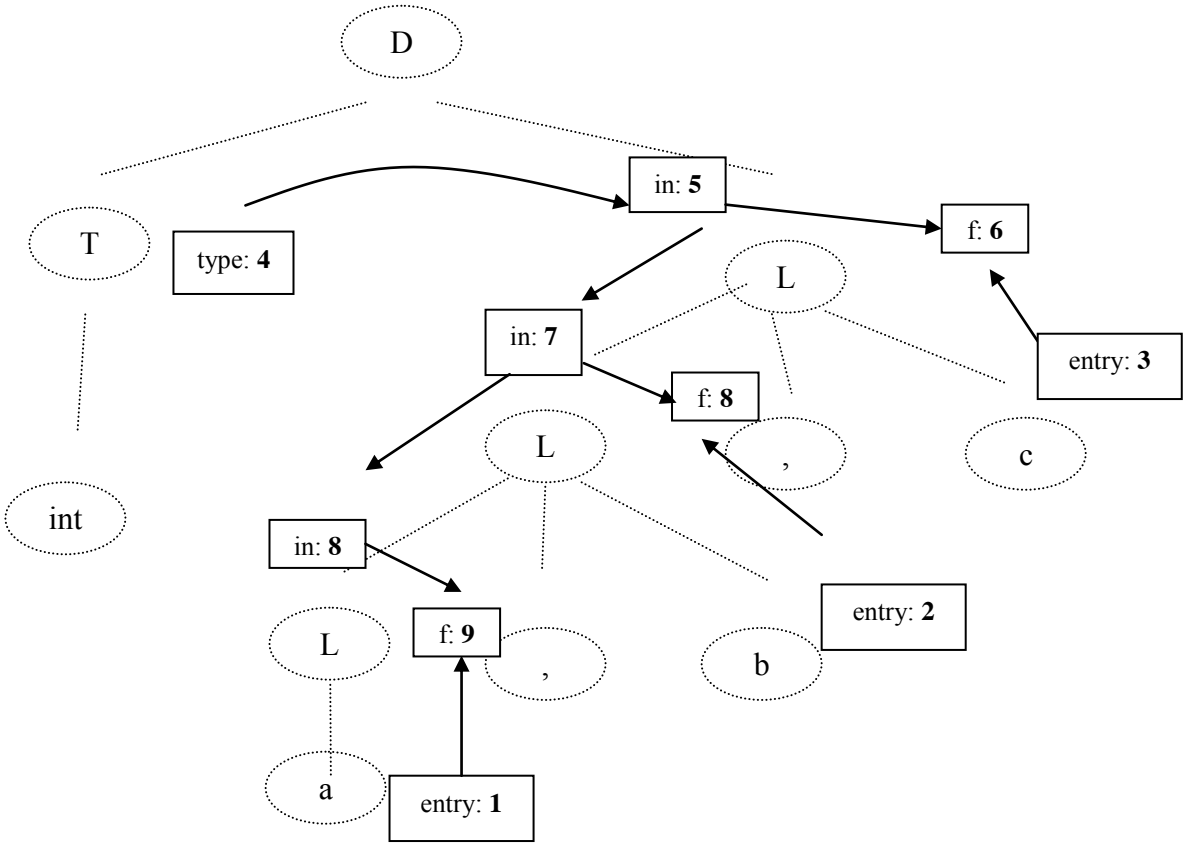
Nút 3: có nhãn là +, con trái là nút 1, con phải là nút 2.

Nút 4: có nhãn là :=, con trái là nút 1, con phải là nút 3.

6. THỨ TỰ ĐÁNH GIÁ THUỘC TÍNH

Trên đồ thị phụ thuộc được xây dựng, ta phải xác định thứ tự của các nút để làm sao cho khi duyệt các nút theo thứ tự này thì một nút sẽ có thứ tự sau nút mà nó phụ thuộc ta gọi là một sắp xếp topo. Tức là nếu các nút được đánh thứ tự m_1, m_2, . . .,m_k thì nếu có m_i ->m_j là một cạnh từ m_i đến m_j thì m_i xuất hiện trước m_j trong thứ tự đó hay i<j. Nếu chúng ta duyệt theo thứ tự đã được sắp xếp này thì sẽ được một cách duyệt hợp lý cho các hành động ngữ nghĩa. Nghĩa là trong một sắp xếp topo, giá trị các thuộc tính phụ thuộc c_1,c_2, . . . ,c_k trong một hành động ngữ nghĩa b:=f(c_1,c_2, . . . ,c_k) đã được tính trước khi ta ước lượng f.

Đối với ví dụ trên, chúng ta xây dựng được một thứ tự phụ thuộc trên các thuộc tính đối với cây cú pháp cho câu vào “int a,b,c” như sau:



Sau khi chúng ta đã có đồ thị phụ thuộc này, chúng ta thực hiện các hành động ngữ nghĩa theo thứ tự như sau (ký hiệu a_i là giá trị thuộc tính ở nút thứ i):

Đối với nút 1,2 ,3 chúng ta duyệt qua nhưng chưa thực hiện hành động ngữ nghĩa nào cả

nút 4: ta có a_4 := real

nút 5: $a_5 := a_4 :=$ real

nút 6: addtype(c.entry,a_5) = addtype(c.entry,real)

nút 7: $a_7 := a_5 :=$ real

nút 8: addtype(b.entry,a_7) = addtype(b.entry,real)

nút 9: addtype(a.entry,a_8) = addtype(a.entry,real)

Các phương pháp duyệt hành động ngữ nghĩa

6.1 Phương pháp dùng cây phân tích cú pháp

- Cây phân tích đã được tạo rồi
- Xây dựng một thứ tự duyệt hay một sắp xếp topo của đồ thị từ cây phân tích cú pháp đó: duyệt cây phân tích từ trái sang phải và theo chiều cao.
- Nếu cần thiết có thể duyệt đi duyệt lại vài lần.
- Phương pháp này không thực hiện được nếu đồ thị phụ thuộc có "vòng".

* Thủ tục duyệt theo chiều sâu

```
procedure dfvisit(n:node);
```

```
begin
        for mỗi con m của n tính từ trái sang phải do
        begin
                tính các thuộc tính kế thừa của m
                dfvisit(m)
        end
        tính các thuộc tính tổng hợp của n
end
```

6.2 Phương pháp dựa trên luật

Vào lúc xây dựng trình biên dịch, các luật ngữ nghĩa được phân tích (thủ công hay bằng công cụ) để thứ tự thực hiện các hành động ngữ nghĩa đi kèm với các sản xuất được xác định trước vào lúc xây dựng.

6.3 Phương pháp quên lãng (oblivious method)

Một thứ tự duyệt được lựa chọn mà không cần xét đến các luật ngữ nghĩa. Thí dụ nếu quá trình dịch xảy ra trong khi phân tích cú pháp thì thứ tự duyệt phải phù hợp với phương pháp phân tích cú pháp, độc lập với luật ngữ nghĩa.

Tuy nhiên phương pháp này chỉ thực hiện trên một lớp các cú pháp điều khiển nhất định.

Trong thực tế, các ngôn ngữ lập trình thông thường có yêu cầu quá trình phân tích là tuyến tính, quá trình phân tích ngữ nghĩa phải kết hợp được với các phương pháp phân tích cú pháp tuyến tính như LL, LR. Để thực hiện được điều này, các thuộc tính ngữ nghĩa cũng cần thoả mãn điều kiện: một thuộc tính ngữ nghĩa sẽ được sinh ra chỉ phụ thuộc vào các thông tin trước nó. Chính vì vậy chúng ta sẽ xét một lớp cú pháp điều khiển rất thông dụng và được sử dụng hiệu quả gọi là cú pháp điều khiển thuần tính L.

Cú pháp điều khiển thuần tính L

Một cú pháp điều khiển gọi là thuần tính L nếu mỗi thuộc tính kế thừa của X_i ở vế phải của luật sinh A -> X_1 X_2 . . . X_n với 1<=j<=n chỉ phụ thuộc vào:

1. Các thuộc tính của các ký hiệu X_1, X_2, . . .,X_{j-1} ở bên trái của X_j trong sản xuất và
2. Các thuộc tính kế thừa của A

Luật sinh	Luật ngữ nghĩa
A → L M	L.i := l(A.i) M.i := m(L.s)

A → Q R	A.s := f(M.s) R.i := r(A.i) Q.i := q(R.s) A.s := f(Q.r)

Ví dụ: Cho định nghĩa trực tiếp cú pháp

Luật sinh	Luật ngữ nghĩa
A → L M	L.i := l(A.i) M.i := m(L.s) A.s := f(M.s)
A → Q R	R.i := r(A.i) Q.i := q(R.s) A.s := f(Q.r)

Đây không phải là một định nghĩa L_thuộc tính vì thuộc tính kế thừa Q.i phụ thuộc vào thuộc tính R.s của ký hiệu bên phải nó trong luật sinh.

Chú ý rằng mỗi cú pháp điều khiển thuần tính S đều thuần tính L vì các điều kiện trên chỉ áp dụng cho các thuộc tính kế thừa.

Phương pháp dựa trên qui tắc và phương pháp quên lãng không nhất thiết phải xây dựng một đồ thị phụ thuộc, vì vậy nó rất là hiệu quả về mặt thời gian cũng như không gian tính toán.

6.4 Đánh giá thuộc tính "trên cùng chuyến bay"

- Phương pháp này, việc đánh giá thuộc tính được thực hiện cùng với bộ phân tích chứ không phải sau nó. Đây là mô hình tốt cho các trình biên dịch duyệt một lần. Khi sử dụng phương pháp này, có hai vấn đề cần quan tâm. Đó là phương pháp phân tích và kiểu thuộc tính được dùng.

Bộ đánh giá thuộc tính-L LL(1)

- Bộ đánh giá dùng một ngăn xếp chứa thuộc tính.
- Khi một ký hiệu không kết thúc được xác định, các thuộc tính kế thừa được đẩy vào ngăn xếp.
- Khi vế phải của sản xuất được xử lý, thì các thuộc tính kế thừa và tổng hợp của mỗi ký hiệu vế phải này cũng được đẩy vào.
- Khi toàn bộ vế phải được xử lý xong thì toàn bộ các thuộc tính của vế phải được lấy ra, và các thuộc tính tổng hợp của vế trái được đẩy vào

Trong thiết kế dịch là dịch một lượt: khi ta đọc đầu vào đến đâu thì chúng ta sẽ phân tích cú pháp đến đó và thực hiện các hành động ngữ nghĩa luôn.

Một phương pháp xây dựng chương trình phân tích cú pháp kết hợp với thực hiện các hành động ngữ nghĩa như sau:

- Với mỗi ký hiệu không kết thúc được gắn với một hàm thực hiện. Giả sử với ký hiệu A, ta có hàm thực hiện void ParseA(Symbol A);
- Mỗi ký hiệu kết thúc được gắn với một hàm đối sánh xâu vào
- Giả sử ký hiệu không kết thúc A là vế trái của luật $A \rightarrow \alpha_1|\alpha_2| \ldots |\alpha_n$

Như vậy hàm phân tích ký hiệu A sẽ được định nghĩa như sau:

```
void ParseA(Symbol A, Rule r, ...)
{if(r==A→ α1)  gọi hàm xử lý ngữ nghĩa tương ứng luật A→α1
else
    if(r==A→ α2) gọi hàm xử lý ngữ nghĩa tương ứng luật A→α2
    . . .
    else
    if(r==A→ αn) gọi hàm xử lý ngữ nghĩa tương ứng luật A→αn
}
```

Đối chiếu ký hiệu đầu vào và A, tìm trong bảng phân tích LL xem sẽ khai triển A theo luật nào. Chẳng hạn ký hiệu xâu vào hiện thời a $\in$ first(α_i), chúng ta sẽ khai triển A theo luật $A \rightarrow X_1 \ldots X_k$ với $\alpha_i = X_1 \ldots X_k$

Ở đây, ta sẽ sử dụng lược đồ dịch để kết hợp phân tích cú pháp và ngữ nghĩa. Do đó đó khi khai triển A theo vế phải, ta sẽ gặp 3 trường hợp sau:

1. Nếu phần tử đang xét là một ký hiệu kết thúc, ta gọi hàm đối sánh với xâu vào, nếu thoả mãn thì nhẩy con trỏ đầu vào lên một bước, nếu trái lại là lỗi.
2. Nếu phần tử đang xét là một ký hiệu không kết thúc, chúng ta gọi hàm duyệt ký hiệu không kết thúc này với tham số bao gồm các thuộc tính của các ký hiệu anh em bên trái, và thuộc tính kế thừa của A.
3. Nếu phần tử đang xét là một hành động ngữ nghĩa, chúng ta thực hiện hành động ngữ nghĩa này.

Ví dụ:

$E \rightarrow T \{R.i:=T.val\}\ R \{E.val:=R.s\}$

$R \rightarrow + T \{R_1.i:=R.i+T.val\}\ R_1 \{R.s:=R_1.s\}$

$R \rightarrow \varepsilon \{R.s:=R.i\}$

$T \rightarrow (E) \{T.val:=E.val\}$

$T \rightarrow num \{T.val:=num.val\}$

```
void ParseE(...)
{       // chỉ có một lược đồ dịch: E → T {R.i:=T.val}  R {E.val:=R.s}
        ParseT(...);   R.i := T.val
        ParseR(...);   E.val := R.s
}
void ParseR(...)
{       // trường hợp 1 R → +T{R1.i:=R.i+T.val}R1 {R.s:=T.val+R1.i}
        if(luật=R→TR1)
        {       match('+');// đối sánh
                ParseT(...);            R1.i:=R.i+T.val;
                ParseR(...);            R.s:=R1.s
        }
        else if(luật=R->ε)
        { // R ->ε{R.s:=R.i}
                R.s:=R.i
        }
 }
```

Tương tự đối với hàm ParseT()

Xét xâu vào: "6+4"

First(E)=First(T) = {(,num} First(R) = {ε,+} Follow(R) = {$,)}

Xây dựng bảng LL(1)

	num	+	(	)	$
E	E →TR		E → TR		
T	T→ num		T → (E)		
R		R → +TR		R → ε	R→ ε

Đầu vào "6+4", sau khi phân tích từ vựng ta được "num1 + num2"

Ngăn xếp	Đầu vào	Luật sản xuất	Luật ngữ nghĩa
$E	num1 + num2 $	E->TR	
$RT	num1 + num2 $	T->num1	T.val=6
$Rnum1	num1 + num2 $		
$R	+ num2 $	R->+TR1	R.i=T.val=6
$R1T+	+ num2 $		

$\$R_1T$	num2 $	T->num2	T.val=4
$\$R_1$num2	num2 $		
$\$R_1$	$	R_1->ε	R_1.i=T.val=4
$	$		R_1.s=T.val+R_1.i=10
			R.s=R_1.s=10
			E.val=R.s=10

Thực hiện hành động ngữ nghĩa trong phân tích LR

Đối với cú pháp điều khiển thuần tính S (chỉ có các thuộc tính tổng hợp), tại mỗi bước thu gọn bởi một luật, chúng ta thực hiện các hành động ngữ nghĩa tính thuộc tính tổng hợp của vế trái dựa vào các thuộc tính tổng hợp của các ký hiệu vế phải đã được tính.

Ví dụ: Cú pháp điều khiển tính giá trị biểu thức cho máy tính bỏ túi:

Luật cú pháp	**Luật ngữ nghĩa (luật dịch)**
L->E n	print(E.val)
E->E_1+T	E.val:=E_1.val+T.val
E->T	E.val:=T.val
T->T_1*F	T.val:=T_1.val*F.val
T->F	T.val:=F.val
F->(E)	F.val:=E.val
F->digit	F.val:=digit.lexval

Ta thực hiện các luật ngữ nghĩa này bằng cách sinh ra thêm một ngăn xếp để lưu giá trị thuộc tính *val* cho các ký hiệu (gọi là ngăn xếp giá trị). Mỗi khi trong ngăn xếp trạng thái có ký hiệu mới, ta đặt vào ngăn xếp giá trị thuộc tính val cho ký hiệu mới này. Còn nếu khi ký hiệu bị loại bỏ ra khỏi ngăn xếp trạng thái thì ta cũng loại bỏ giá trị tương ứng với nó ra khỏi ngăn xếp giá trị.

Ví dụ: Xem quá trình phân tích gạt, thu gọn với xâu vào "3*5+4":

+ Với ký hiệu không có giá trị *val*, ta ký hiệu '-' cho *val* của nó

xâu vào	**ngăn xếp trạng thái**	**ngăn xếp giá trị**	**Luật cú pháp, ngữ nghĩa**
d1 * d2 + d3 n			gạt
* d2 + d3 n	d1	3	F→digit
* d2 + d3 n	F	3	F.val:=digit.lexval (loại bỏ digit)

			T→F
* d2 + d3 n	T	3	T.val:=F.val (loại bỏ F) gạt
d2 + d3 n	* T	- 3	gạt
+ d3 n	d2 * T	5 – 3	F→digit
+ d3 n	F * T	5 – 3	F.val:=digit.lexval *(loại bỏ digit)* $T \rightarrow T_1 * F$
+ d3 n	T	15	$T.val:=T_1.val*F.val$ (loại bỏ T_1,*,F) E→T
+ d3 n	E	15	E.val:=T.val (loại bỏ T) gạt
d3 n	+ E	- 15	gạt
n	d3 + E	4 – 15	F→digit
n	F + E	4 – 15	F.val:=digit.lexval (loại bỏ digit) T→F
n	T + E	4 – 15	T.val:=F.val (loại bỏ F) $E \rightarrow E_1+T$
n	E	19	$E.val:=E_1.val+T.val$ (loại bỏ E_1,+,T) gạt
	E n	- 19	L→E n
	L	19	L.val:=E.val (loại bỏ E,n)

Chú ý là không phải mọi cú pháp điều khiển thuần tính L đều có thể kết hợp thực hiện các hành động ngữ nghĩa khi phân tích cú pháp mà không cần xây dựng cây cú pháp. Chỉ có một lớp hạn chế các cú pháp điều khiển có thể thực hiện như vậy, trong đó rõ nhất là cú pháp điều khiển thuần tuý S.

7. LƯỢC ĐỒ CHUYỂN ĐỔI(Lược đồ dịch) - Translation Scheme

Nếu mô tả các hành động ngữ nghĩa theo cú pháp điều khiển thì không xác định thứ tự của các hành động trong một sản xuất. Vì vậy ở đây ta xét một tiếp cận khác là dùng lược đồ dịch để mô tả luật ngữ nghĩa đồng thời với thứ tự thực hiện chúng trong một sản xuất.

Lược đồ chuyển đổi là một văn phạm phi ngữ cảnh trong đó các thuộc tính được liên kết với các ký hiệu văn phạm và các hành động ngữ nghĩa nằm giữa hai dấu ngoặc móc {} được chèn vào một vị trí nào đó bên vế phải của sản xuất.

+ Lược đồ dịch vẫn có cả thuộc tính tổng hợp và thuộc tính kế thừa

+ Lược đồ dịch xác định thứ tự thực hiện hành động ngữ nghĩa trong mỗi sản xuất

Ví dụ: một lược đồ dịch để sinh biểu thức hậu vị cho một biểu thức như sau:

E -> T R

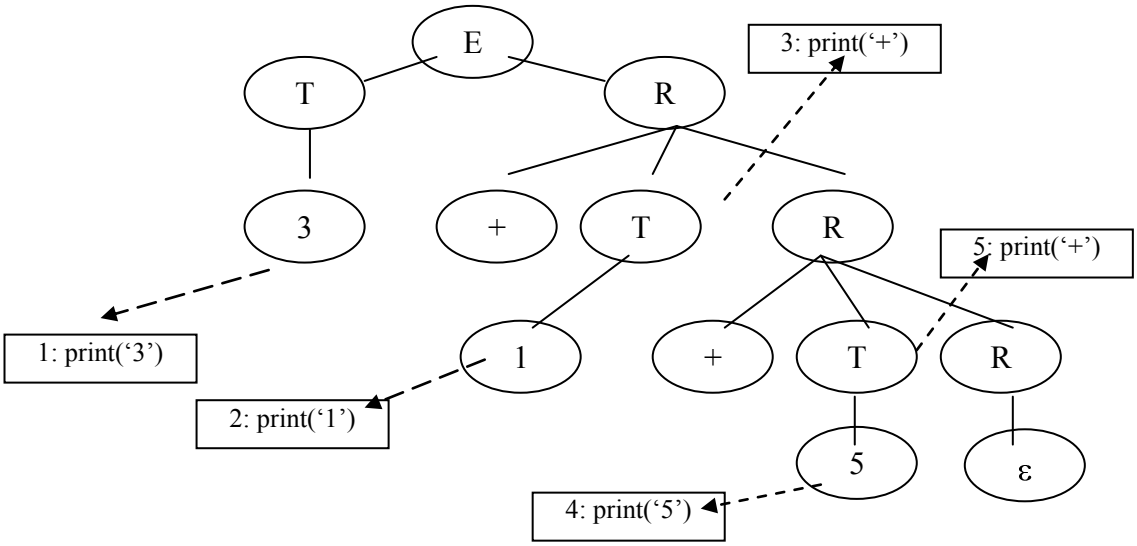
R -> + T *{print('+')}* R

R -> ε

T -> num *{print(num.val)}*

Xét biểu thức "3+1+5"

Dyệt theo thủ tục duyệt theo chiều sâu, các hành động ngữ nghĩa được đánh thứ tự lần lượt 1,2,3, . . .



Kết quả dịch là "3 1 + 5 +"

Chú ý là nếu trong lược đồ dịch ta đặt hành động ngữ nghĩa ở vị trí khác đi, chúng ta sẽ có kết quả dịch khác ngay. Ví dụ, đối với lược đồ dịch, ta thay đổi một chút thành lược đồ dịch như sau:

E -> T R

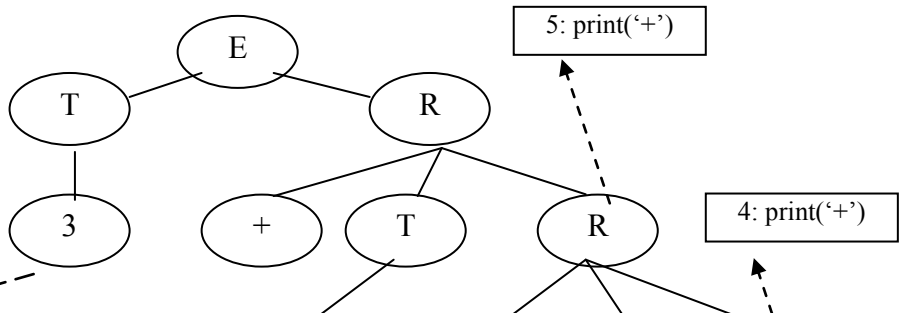
R -> + T R *{print('+')}*

R -> ε

T -> num *{print(num.val)}*

Xét biểu thức "3+1+5"

Duyệt theo thủ tục duyệt theo chiều sâu, các hành động ngữ nghĩa được đánh thứ tự lần lượt 1,2,3, . . .



Kết quả dịch là "3 1 5 + +"

Khi thiết kế lược đồ dịch, chúng ta cần một số điều kiện để đảm bảo rằng một giá trị thuộc tính phải có sẵn khi chúng ta tham chiếu đến nó:

1. Một thuộc tính kế thừa cho một ký hiệu ở vế phải của một sản xuất phải được tính ở một hành động nằm trước ký hiệu đó.
2. Một hành động không được tham chiếu đến thuộc tính của một ký hiệu ở bên phải của hành động đó.
3. Một thuộc tính tổng hợp cho một ký hiệu không kết thúc ở vế trái chỉ có thể được tính sau khi tất cả thuộc tính nó cần tham chiếu đến đã được tính xong. Hành động như thế thường được đặt ở cuối vế phải của luật sinh.

Ví dụ lược đồ dịch sau đây không thoả mãn các yêu cầu này:

S -> A_1 A_2 *{A_1.in:=1; A_2.in:=2}*

A -> a *{print(A.in)}*

Ta thấy thuộc tính kế thừa A.in trong luật thứ 2 chưa được định nghĩa vào lúc muốn in ra giá trị của nó khi duyệt theo hướng sâu trên cây phân tích cho đầu vào aa. Để thoả mãn thuộc tính L, chúng ta có thể sửa lại lược đồ trên thành như sau:

S -> {A_1.in:=1 } A_1 {A_2.in:=2} A_2

A -> a {print(A.in)}

Như vậy, thuộc tính A.in được tính trước khi chúng ta duyệt A.

Những điều kiện này được thoả nếu văn phạm có điều khiển thuần tính L. Khi đó chúng ta sẽ đặt các hành động theo nguyên tắc như sau:

1. Hành động tính thuộc tính kế thừa của một ký hiệu văn phạm A bên vế phải được đặt trước A.

2. Hành động tính thuộc tính tổng hợp của ký hiệu vế trái được đặt ở cuối luật sản xuất.

Ví dụ:

Cho văn phạm biểu diễn biểu thức gồm các toán tử + và - với toán hạng là các số:

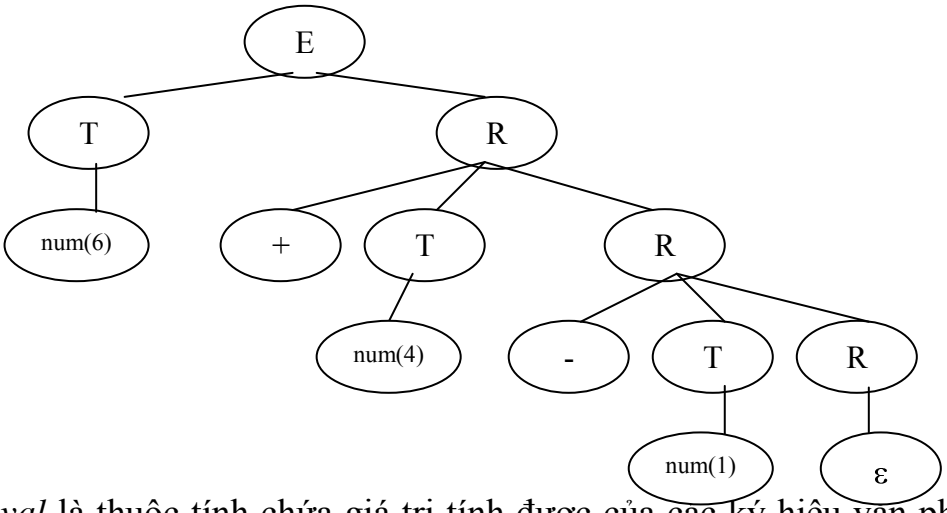
$E \rightarrow T\ R$ $R \rightarrow +\ T\ R$ $R \rightarrow -\ T\ R$

$R \rightarrow \varepsilon$ $T \rightarrow (\ E\ )$ $T \rightarrow num$

Xây dựng lược đồ dịch trên văn phạm này để tính giá trị của biểu thức.

Giải đáp:

Trước hết, chúng ta thử xem cây phân tích cú pháp cho đầu vào "6+4-1"



Gọi *val* là thuộc tính chứa giá trị tính được của các ký hiệu văn phạm E và T. Thuộc tính *s* là thuộc tính tổng hợp và *i* là thuộc tính kế thừa để chứa giá trị tính được của ký hiệu R. Ta đặt R.i chứa giá trị của phần biểu thức đứng trước R và R.s chứa kết quả. Ta xây dựng lược đồ dịch như sau:

$E \rightarrow T\ \{R.i:=T.val\}\ R\ \{E.val:=R.s\}$

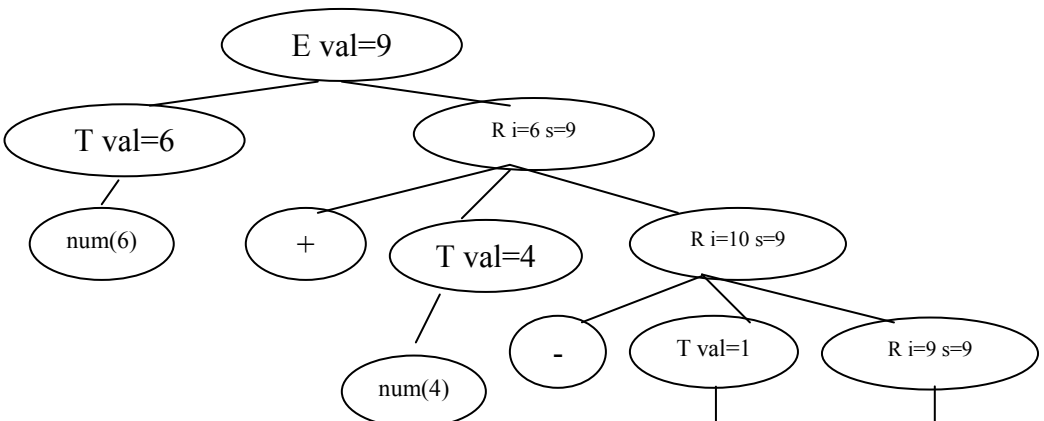
$R \rightarrow +T\ \{R_1.i:=R.i+T.val\}R_1\ \{R.s:=R_1.s\ \}$

$R \rightarrow -T\ \{\ R_1.i:=R.i-T.val\ \}R_1\ \{R.s:=R_1.s\}$

$R \rightarrow \varepsilon\ \{R.s:=R.i\}$

$T \rightarrow (\ E\ )\ \{T.val:=E.val\}$

$T \rightarrow num\ \{T.val:=num.val\}$



Lưu ý:

Nếu chúng ta xác định một cách duyệt khác cách duyệt theo hướng sâu thì cách đặt hành động dịch vào vị trí nào sẽ được làm khác đi. Tuy nhiên cách duyệt theo hướng sâu là cách duyệt phổ biến nhất và tự nhiên nhất (vì ngữ nghĩa sẽ được xác định dần theo chiều duyệt đầu vào từ trái sang phải) nên chúng ta coi khi duyệt một cây phân tích, chúng ta sẽ duyệt theo hướng sâu.

BÀI TẬP

Câu 1:

Cho cú pháp điều khiển tính giá trị của một số ở hệ cơ số hai như sau:

Luật cú pháp	**Luật ngữ nghĩa**
B→ 0	B.val=0;
B→1	B.val:=1;
$B \rightarrow B_1$ 0	B.val:=2*B_1.val +0
$B \rightarrow B_1$ 1	B.val:=2*B_1.val+1
B→0	B.val=0;
B→1	B.val:=1;

Hãy xây dựng cây phân tích cú pháp đối với xâu vào "1010" và thực hiện các luật ngữ nghĩa trên cây phân tích này theo chiều sâu (chỉ ra thứ tự các bước thực hiện luật ngữ nghĩa).

Câu 2:

Cho văn phạm sau mô tả một số khai báo biến trong ngôn ngữ C như sau: (với D, T, L là ký hiệu không kết thúc và D là ký hiệu bắt đầu).

D -> T L

L -> id | id , L | id [num] | id [num] , L

T -> int | float

Hãy xây dựng lược đồ dịch trên các sản xuất văn phạm đấy để sinh kiểu cho các biến. Xây dựng cây phân tích và thực hiện các luật ngữ nghĩa theo lược đồ dịch trên cây phân tích đó với xâu vào "float a, b[3]" để tính kiểu của a và b.

Câu 3:

Tương tự bài 2 nhưng thay lược đồ dịch bằng cú pháp điều khiển.

CHƯƠNG 5
PHÂN TÍCH NGỮ NGHĨA

Mục tiêu: Cần nắm được:

- *Hệ thống kiểu với các biểu thức kiểu thường gặp ở các ngôn ngữ lập trình.*
- *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*

Nội dung:

Nhiệm vụ của giai đoạn kiểm tra ngữ nghĩa là kiểm tra tính đúng đắn về mặt ngữ nghĩa của chương trình nguồn. Việc kiểm tra được chia làm hai loại là kiểm tra tĩnh và kiểm tra động (Việc kiểm tra của chương trình dịch được gọi là tĩnh, việc kiểm tra thực hiện trong khi chương trình đích chạy gọi là động. Một kiểu hệ thống đúng đắn sẽ xoá bỏ sự cần thiết kiểm tra động).

Có một số dạng của kiểm tra tĩnh:

- Kiểm tra kiểu: kiểm tra về tính đúng đắn của các kiểu toán hạng trong biểu thức.

- Kiểm tra dòng điều khiển: một số điều khiển phải có cấu trúc hợp lý, ví dụ như lệnh break trong ngôn ngữ pascal phải nằm trong một vòng lặp.

- Kiểm tra tính nhất quán: có những ngữ cảnh mà trong đó một đối tượng được định nghĩa chỉ đúng một lần. Ví dụ, trong Pascal, một tên phải được khai báo duy nhất, các nhãn trong lệnh case phải khác nhau, và các phần tử trong kiểu vô hướng không được lặp lại.

- Kiểm tra quan hệ tên: Đôi khi một tên phải xuất hiện từ hai lần trở lên. Ví dụ, trong Assembly, một chương trình con có một tên mà chúng phải xuất hiện ở đầu và cuối của chương trình con này.

Nội dung trong phần này, ta chỉ xét một số dạng trong kiểm tra kiểu của chương trình nguồn.

1. BIỂU THỨC KIỂU (type expressions)

Kiểu của một cấu trúc ngôn ngữ được biểu thị bởi "biểu thức kiểu". Một biểu thức kiểu có thể là một kiểu cơ bản hoặc được xây dựng từ các kiểu cơ bản theo một số toán tử nào đó.

Ta xét một lớp các biểu thức kiểu như sau:

1) <u>Kiểu cơ bản:</u>

Gồm *boolean, char, interger, real*. Có các kiểu cơ bản đặc biệt là *type_error* (để trả về một cấu trúc bị lỗi kiểu), *void* (biểu thị các cấu trúc không cần xác định kiểu như câu lệnh).

2) Kiểu hợp thành:

+ *Mảng:* Nếu *T* là một biểu thức kiểu thì *array(I,T)* là một biểu thức kiểu đối với một mảng các phần tử kiểu *T* và *I* là tập các chỉ số.

Ví dụ: trong ngôn ngữ Pascal khai báo: var A: array[1..10] of interger;

sẽ xác định kiểu của A là array(1..10,interger)

+ *Tích của biểu thức kiểu:* là một biểu thức kiểu. Nếu T_1 và T_2 là các kiểu biểu thức kiểu thì tích Đề các của $T_1 x T_2$ là một biểu thức kiểu.

+ *Bản ghi*: Kiểu của một bản ghi chính là biểu thức kiểu được xây dựng từ các kiểu của các trường của nó.

Ví dụ trong ngôn ngữ Pascal:

```
type row=record
        address: interger;
        lexeme: array[1..15] of char;
end;
var table: array[1..101] of row;
```

Như vậy một biến của row thì tương ứng với một biểu thức kiểu là:

record((address x interger) x (lexeme x array(1..15,char)))

+ *Con trỏ:* Giả sử *T* là một biểu thức kiểu thì *pointer(T)* là một biểu thị một biểu thức kiểu xác định kiểu cho con trỏ của một đối tượng kiểu *T*.

Ví dụ, trong ngôn ngữ Pascal: var p: ^row thì p có kiểu là pointer(row)

+ *Hàm:* Một hàm là một ánh xạ từ các phần tử của một tập vào một tập khác. Kiểu một hàm là ánh xạ từ một kiểu miền D vào một kiểu phạm vi R. Biểu thức kiểu cho một hàm như vậy sẽ được ký hiệu là D → R.

Ví dụ trong ngôn ngữ Pascal, một hàm khai báo như sau:

```
function f(a,b:interger): ^interger;
```

có kiểu miền là interger x interger và kiểu phạm vi là pointer(interger). Và như vậy biểu thức kiểu xác định kiểu cho hàm đó là:

interger x interger → pointer(interger)

2 CÁC HỆ THỐNG KIỂU

Một hệ thống kiểu là một tập các luật để xác định kiểu cho các phần trong chương trình nguồn. Một bộ kiểm tra kiểu làm nhiệm vụ thực thi các luật trong hệ thống kiểu này. Ở đây, hệ thống kiểu được xác định bởi các luật ngữ nghĩa dựa trên luật cú pháp.

Một hệ thống kiểu đúng đắn sẽ xoá bỏ sự cần thiết phải kiểm tra động (vì nó cho phép xác định tĩnh, các lỗi không xảy ra trong lúc chương trình đích chạy). Một ngôn ngữ gọi là định kiểu mạnh nếu chương trình dịch của nó có thể bảo đảm rằng các chương trình mà nó dịch tốt sẽ hoạt động không có lỗi về kiểu. Điều quan trọng là khi bộ kiểm tra phát hiện lỗi, nó phải khắc phục lỗi dể tiếp tục kiểm tra. trước hết nó thông báo về lỗi mô tả và vị trí lỗi. Lỗi xuất hiện gây ảnh hưởng đến các luật kiểm tra lỗi, do vậy phải thiết kế kiểu hệ thống như thế nào để các luật có thể đương đầu với các lỗi này.

Đối với câu lệnh không có giá trị, ta gán cho nó kiểu cơ sở đặc biệt void. Nếu có lỗi về kiểu thì ta gán cho nó giá trị kiểu là type_error

Xét cách xây dựng luật ngữ nghĩa kiểm tra kiểu qua một số ví dụ sau:

Ví dụ 1: Văn phạm cho khai báo:

D → D ; D

D → id : T

T → interger| char| ^ T| array [num] of T| boolean| real

Luật cú pháp	**Luật ngữ nghĩa**
D → id : T	AddType(id.entry,T.type)
T → char	T.type := char
T → interger	T.type := interger
T → ^T_1	T.type := pointer(T_1.type)
T → array [num] of T_1	T.type := array(num.val,T_1.type)
T → real	T.type := real
T → boolean	T.type := boolean

Hành động ứng với sản xuất D → **Tên**:T lưu vào bảng kí hiệu một kiểu cho một tên. Hàm {addtype (tên.entry, T.type)} nghĩa là cất một thuộc tính T.type vào bản kí hiệu ở vị trí entry.

Ví dụ 2: Văn phạm sau cho biểu thức

S → id := E

E → E + E| E mod E| E_1 [E_2]| num| id| letter

Luật cú pháp	**Luật ngữ nghĩa**
S -> id := E	S.type := if id.type=E.type then void else type_error
E -> E_1 + E_2	E.type:= if E_1.type=interger and E_2.type=interger then interger else if E_1.type=interger and E_2.type=real then real else if E_1.type=real and E_2.type=interger then real else if E_1.type=real and E_2.type=real then real else type_error
E -> num	E.type := interger
E -> id	E.type := GetType(id. Entry)
E -> E_1 mod E_2	E.type := **if** E_1.type=interger **and** E_2.type=interger **then** interger **else** type_error
E -> E_1 [E_2]	E.type := **if** E_2.type=interger **and** E_1.type=array(s,t) **then** t **else** type_error
E → letter	E.type := char

Ví dụ 3: Kiểm tra kiểu cho các câu lệnh:

S → if E then S | while E do S| S_1 ; S_2

Luật cú pháp	**Luật ngữ nghĩa**
S -> if E then S_1	S.type := **if** E.type=boolean **then** S_1.type **else** type_error
S -> while E do S_1	S.type := **if** E.type=boolean **then** S_1.type **else** type_error
S -> S_1 ; S_2	S.type := **if** S_1.type=void **and** S_2.type=void **then** void **else** type_error

Ví dụ 4: Kiểu hàm: Luật cú pháp thể hiện lời gọi hàm: E → E_1(E_2)

```
function f(a,b:char):^interger;
begin
        . . .
end;
var     p:^interger; q:^char;
        x,y:interger;
begin
. . .
p:=f(x,y);// đúng
q:=f(x,y);// sai
end;
```

Luật cú pháp	Luật ngữ nghĩa
E -> E_1 (E_2)	E.type := **if** E_2.type=s **and** E_1.type=s->t **then** t **else** type_error

3 MỘT SỐ VẤN ĐỀ KHÁC CỦA KIỂM TRA KIỂU

3.1 Sự tương đương của kiểu biểu thức

Nhiều luật có dạng "if kiểu của 2 biểu thức giống nhau thì trả về kiểu đó else trả về type _error" Vậy làn sao để xác định chính xác khi nào thì 2 kiểu biểu thức là tương đương?

Hàm dùng để kiểm tra sự tương đương về cấu trúc của kiểu biểu thức.

```
Function sequiv(s,t): boolean;
begin
if s và t cùng kiểu cơ sở then  return true;
else if s = array (s1,s2) and t = array (t1,t2) then return sequiv(s1,t1) and sequiv(s2,t2)
   else if s=pointer(s1) and t=pointer(t1) then    return sequiv(s1,t1)
        else if s=s1 → s2 and t = t1 → t2 then return sequiv(s1,t1) and sequiv(s2,t2)
             else return false;
end;
```

3.2 Đổi kiểu

Xét biểu thức dạng : x+i, (x: kiểu real, i kiểu integer)

Biểu diễn real và integer trong máy tính là khác nhau đồng thời cách thực hiện phép cộng đối với số real và số integer khác nhau. Để thực hiện phép cộng, trước tiên chương trình dịch đổi 2 toán tử về một kiểu (kiểu real) sau đó thực hiện cộng.

Bộ kiểm tra kiểu trong chương trình dịch được dùng để chèn thêm phép toán vào các biểu diễn trung gian của chương trình nguồn.

Ví dụ: chèn thêm phép toán inttoreal (dùng chuyển một số integer thành số real) rồi mới thực hiện phép cộng số thực real + như sau: xi inttoreal real +

*** Ép kiểu:**

Một phép đổi kiểu được gọi là không rõ (ẩn) nếu nó thực hiện một cách tự động bởi chương trình dịch, phép đổi kiểu này còn gọi là ép kiểu.

Một phép đổi kiểu được gọi là rõ nếu người lập trình phải viết số thứ để thực hiện phép đổi này.

Sản xuất	**Luật ngữ nghĩa**
E → Số	E.type:= integer
E → Số.số	E.type:= real
E → tên	E.type:= lookup (tên.entry)
E → E_1 op E_2	E.type:= if E_1.type = integer and E_2.type = integer Then integer Else if E_1.type = integer and E_2.type = real Then real Else if E_1.type = real and E_2.type = integer Then real Else if E_1.type = real and E_2.type = real Then real **Else type_error**

3.3 Định nghĩa chồng của hàm và các phép toán

Kí hiệu chồng là kí hiệu có nhiều nghĩa khác nhau phụ thuộc vào ngữ cảnh của nó. Ví dụ: + là toán tử chồng, A+B ý nghĩa khác nhau đối với từng trường hợp A,B là số nguyên, số thực, số phức, ma trận…

Định nghĩa chồng cho phép tạo ra nhiều hàm khác nhau nhưng có cùng một tên. Để xác định thực sự dùng định nghĩa chồng nào ta phải căn cứ vào ngữ cảnh lúc áp dụng.

Điều kiện để thực hiện toán tử chồng là phải có sự khác nhau về kiểu hoặc số tham số. Do đó ta có thể dựa vào luật ngữ nghĩa để kiểm tra kiểu và gọi các hàm xử lý.

BÀI TẬP

Câu1:

Viết biểu thức kiểu cho các kiểu sau:

a) Mảng con trỏ trỏ tới số thực, chỉ số mảng từ 1 đến 100

b) Mảng hai chiều với các dòng có chỉ số từ 0 đến 9, cột từ 0 đến 10.

Câu 2:

Cho văn phạm:

$P \rightarrow D ; S$

$D \rightarrow D ; D \mid id : T$

$T \rightarrow integer \mid boolean \mid array [num] of T$

$S \rightarrow S ; S$

$S \rightarrow E := E \mid if\ E\ then\ S \mid$

$E \rightarrow num \mid id \mid E\ mod\ E \mid E [E]$

Trong đó P là ký hiệu bắt đầu; P, D, S, T, E là các ký hiệu không kết thúc; ‘;’, ‘:’, id, integer, boolean, array, num, of, :=, if, then, mod, [,] là các ký hiệu kết thúc; integer là kiểu nguyên, boolean là kiểu logic (giống trong Pascal); id là từ tố tên, num là từ tố số nguyên.

Bạn hãy:

a) Viết lược đồ dịch kiểm tra kiểu cho văn phạm trên

b) Tính kiểu cho các biểu thức trong đoạn chương trình sau:

Câu3

Cho các khai báo sau:

```
a: array[10] of interger;
b: boolean;
a[0] := b
```

Yêu cầu bạn vẽ cây cú pháp ứng với đoạn chương trình trên, sau đó tính kiểu cho các nút trên cây cú pháp đó.

CHƯƠNG 6
BẢNG KÍ HIỆU

Mục đích: Học song chương này sinh viên nắm được:

- *Cách xây dựng bảng kí hiệu.*
- *Biết chọn cấu trúc dữ liệu pù hợp để xây dựng bảng kí hiệu*

Nội dung:

- *Các yêu cầu của bảng kí hiệu*
- *Các cấu trúc dữ liệu dùng để xây dưng bảng kí hiệu*

1. MỤC ĐÍCH, NHIỆM VỤ

Một chương trình dịch cần phải thu thập và sử dụng các thông tin về các tên trong chương trình nguồn. Các thông tin này được lưu trong một cấu trúc dữ liệu gọi là một bảng kí hiệu. Các thông tin bao gồm tên, kiểu, dạng của nó (một biến hay là một cấu trúc), vị trí cảu nó trong bộ nhớ, các thuộc tính khác phụ thuộc vào ngôn gnữ lập trình.

Mỗi lần tên cần xem xét, chương trình dịch sẽ tìm trong bảng kí hiệu xem đã có tên đó chưa. Nếu tên đó là mớithì thêm vào bảng. Các thông tin về tên được tìm và đưa vào bảng trong giai đoạn phân tích từ vựng và cú pháp.

Các thông tin trong bảng kí hiệu được dùng trong:

+ 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

2. CÁC YÊU CẦU ĐỐI VỚI BẢNG KÍ HIỆU

Ta cần có một số khả năng làm viếc với bảng như sau:

1) phát hiện một tên cho trước có trong bảng hay không?

2) thêm tên mới.

3) lấy thông tin tương ứng với tên cho trước.

4) thêm thông tin mới vào tên cho trước.

5) xoá một tên hoặc nhóm tên.

Các thông tin trong bảng kí hiệu có thể gồm:

1) Xâu kí tự tạo nên tên.

2) Thuộc tính của tên.

3) các tham số như số chiều của mảng.

4) Có thể có con trỏ đên tên cấp phát.

Các thông tin đưa vào bảgn trong những thời điểm khác nhau.

3. CẤU TRÚC DỮ LIỆU CỦA BẢNG KÍ KIỆU

Có nhiều cách tổ chức bảng kí hiệu khác nhau như có thể tách bảng riêng rẽ ứng với tên biến, nhãn, hằng số, tên hàm và các kiểu tên khác… tuỳ thuộc vào từng ngôn ngữ.

Về cách tổ chức dữ liệu có thể tỏ chức bởi danh sách tuyến tính, cây tìm kiếm, bảng băm…

Mỗi ô trong bảng ký hiệu tương ứng với một tên. Định dạng của các ô này thường không giống nhau vì thông tin lưu trữ về một tên phụ thuộc vào việc sử dụng tên đó. Thông thường một ô được cài đặt bởi một mẩu tin có dạng (tên, thuộc tính).

Nếu muốn có được sự đồng nhất của các mẩu tin ta có thể lưu thông tin bên ngoài bảng ký hiệu, trong mỗi ô của bảng chỉ chứa các con trỏ trỏ tới thông tin đó.

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

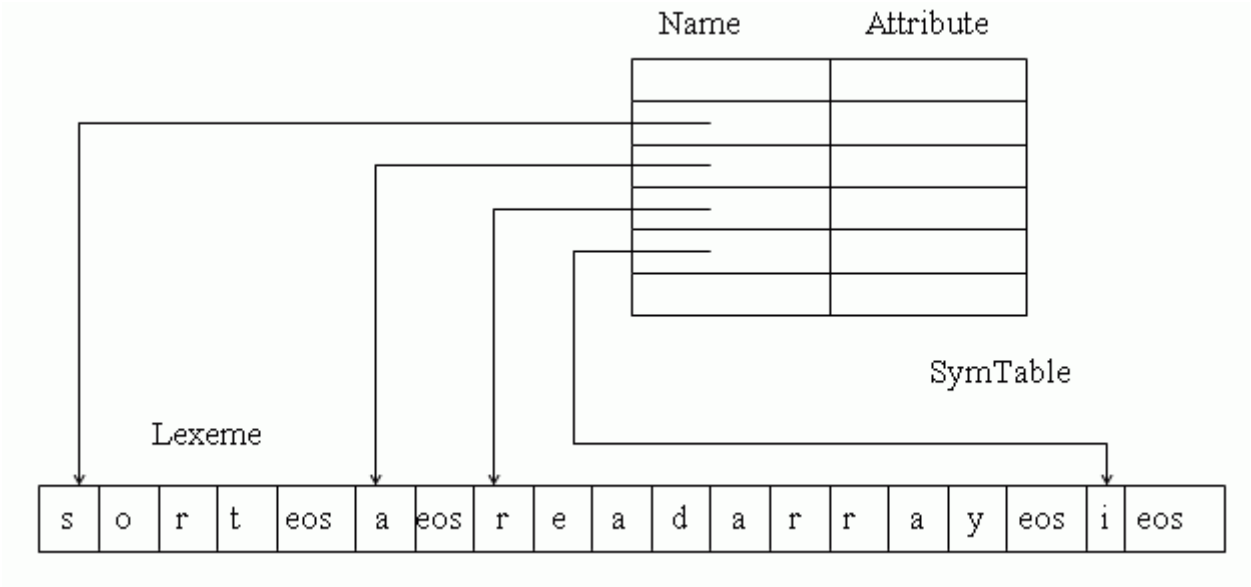
* Nếu ghi trực tiếp tên trong trường tên của bảng thì: ưu điểm: đơn giản, nhanh. Nhược điểm: Độ dài tên bị giới hạn bởi kích thước của trường , hiệu quả sử dụng bộ nhớ không cao.

Trường hợp danh biểu bị giới hạn về độ dài thì chuỗi các ký tự tạo nên danh biểu được lưu trữ trong bảng ký hiệu.

Name									Attribute
s	o	r	t						
a									
r	e	a	d	a	r	r	a	y	
i									

Hình 6.1 Bảng ký hiệu lưu giữ các tên bị giới hạn độ dài

Trường hợp độ dài tên không bị giới hạn thì các Lexeme được lưu trong một mảng riêng, bảng ký hiệu chỉ giữ các con trỏ trỏ tới đầu mỗi Lexeme



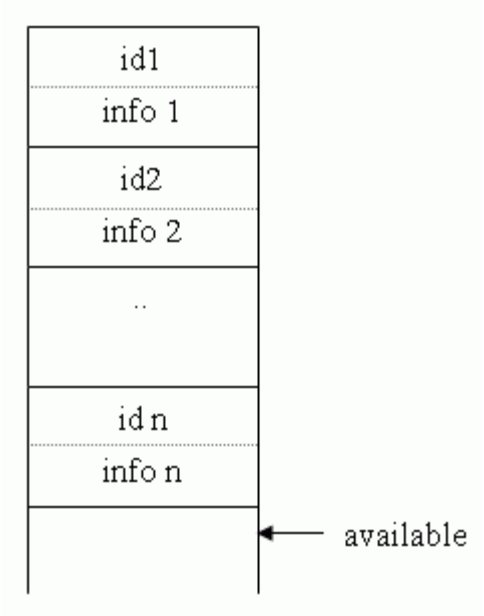
Hình 6.2 Bảng ký hiệu lưu giữ các tên không bị giới hạn độ dài

3.1 Danh sách.

Cấu trúc đơn giản, dễ cài đặt nhất cho bảng ký hiệu là danh sách tuyến tính của các mẩu tin.

Ta dùng một mảng hoặc nhiều mảng tương đương để lưu trữ tên và các thông tin kết hợp với chúng. Các tên mới được đưa vào trong danh sách theo thứ tự mà chúng được phát hiện. Vị trí của mảng được đánh dấu bởi con trỏ available chỉ ra một ô mới của bảng sẽ được tạo ra.

Việc tìm kiếm một tên trong bảng ký hiệu được bắt đầu từ available đến đầu bảng. Trong các ngôn ngữ cấu trúc khối sử dụng quy tắc tầm tĩnh. Thông tin kết hợp với tên có thể bao gồm cả thông tin về độ sâu của tên. Bằng cách tìm kiếm từ available trở về đầu mảng chúng ta đảm bảo rằng sẽ tìm thấy tên trong tầng gần nhất.



Hình 6.3 Danh sách tuyến tính các mẩu tin

3.2 Cây tìm kiếm

Một trong các dạng cây tìm kiếm hiệu quả là: cây tìm kiếm nhị phân tìm kiếm. Các nút của cây có khoá là tên của bản ghi, hai con tro Left, right.

Đối với mọi nút trên cây phải thoả mãn:

- Mọi khoá thuộc cây con trái nhỏ hơn khoá của gốc.

- Mọi nút của cây con phải lớn hơn khoá của gốc.

Giải thuật tìm kiếm trên cây nhị phân:

- So sánh giá trị tìm kiếm x với khoá của gốc:

+ Nếu trùng: tìm kiếm thoả mãn.

+ Nếu < hơn: Thực hiện lại cách tìm kiểm với cây con bên trái.

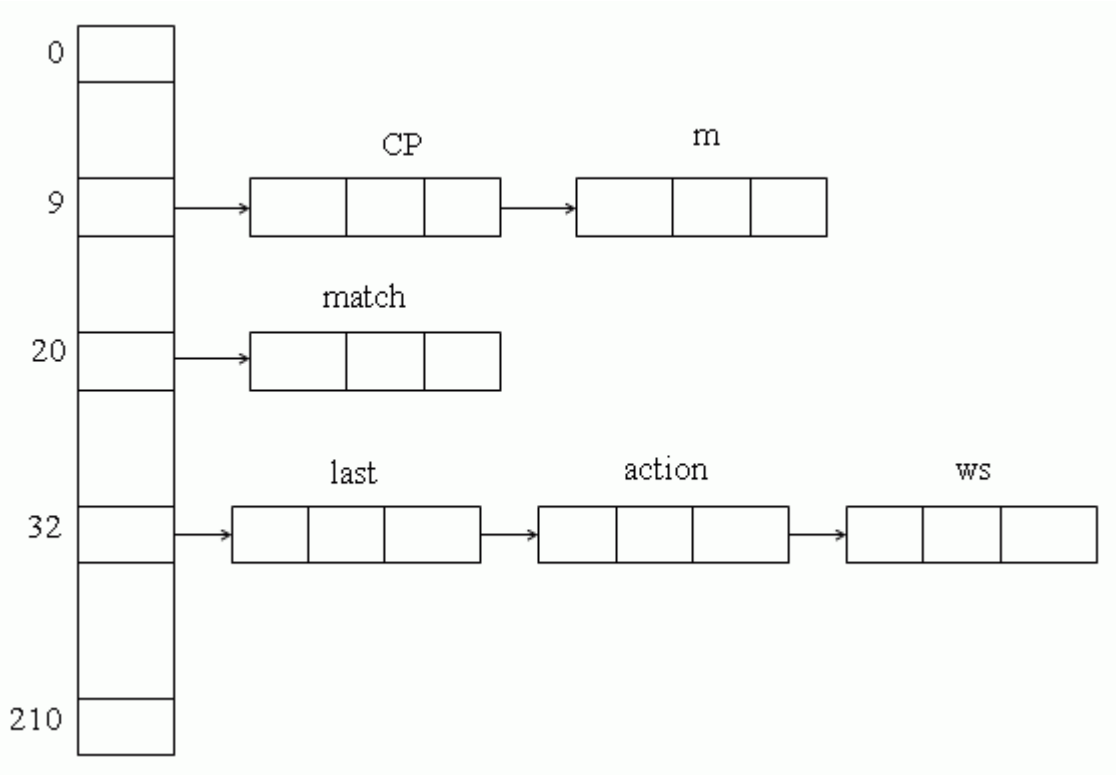
+ Nếu > gốc: thực hiện lại cách tìm kiếm với cây con bên phải.

Để đảm bảo thời gian tìm kiếm người ta thay thé cây nhị phân tìm kiếm bằng cây nhị phân cân bằng.

3.3 Bảng Băm

Kỹ thuật sử dụng bảng băm để cài đặt bảng ký hiệu thường được sử dụng vì tính hiệu quả của nó.

Cấu tạo bao gồm hai phần; bảng băm và các danh sách liên kết.



Hình 6.4 Bảng băm có kích thước 211

1. Bảng băm là một mảng bao gồm m con trỏ.

2. Bảng danh biểu được chia thành m danh sách liên kết, mỗi danh sách liên kết được trỏ bởi một phần tử trong bảng băm.

Việc phân bổ các danh biểu vào danh sách liên kết nào do hàm băm (hash function) quy định. Giả sử s là chuỗi ký tự xác định danh biểu, hàm băm h tác động lên s trả về một giá trị nằm giữa 0 và m- 1 h(s) = t => Danh biểu s được đưa vào trong danh sách liên kết được trỏ bởi phần tử t của bảng.

Có nhiều phương pháp để xác định hàm băm. Phương pháp đơn giản nhất như sau:

1. Giả sử s bao gồm các ký tự c1, c2, c3, ..., ck. Mỗi ký tự cho ứng với một số nguyên dương n1, n2, n3,...,nk; lấy h = n1 + n2 +...+ nk.

2. Xác định h(s) = h mod m

CHƯƠNG 7 SINH MÃ TRUNG GIAN

Mục tiêu:

Sinh viên nắm được cách tạo ra một bộ sinh mã trung gian cho một ngôn ngữ lập trình đơn giản

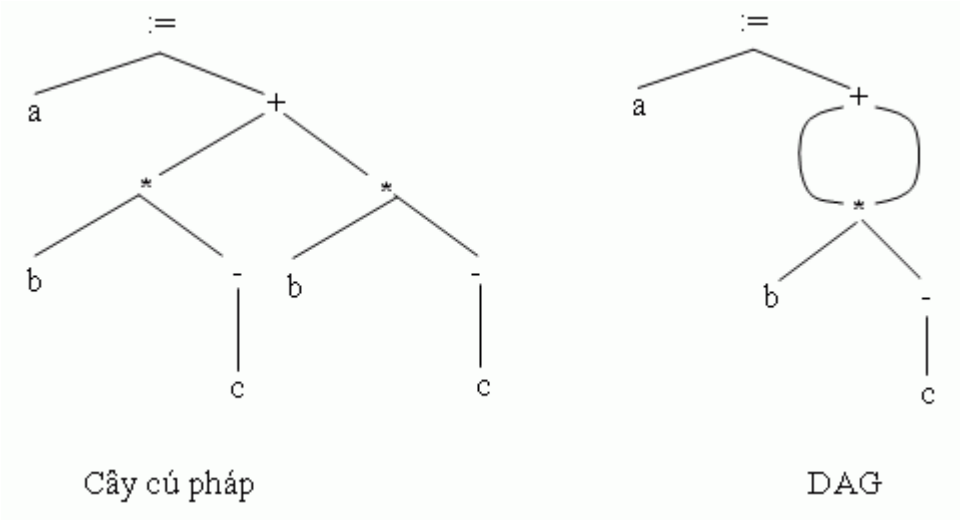
Nội dung:

Các cách biểu diễn mã ở dạng trung gian , đặc biệt là mã ba địa chỉ. Bộ sinh mã ba dịa chỉ dùng định nghĩa trực tiếp cú pháp để dịch các khai báo, âu lệnh sang mã ba địa chỉ

1 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.

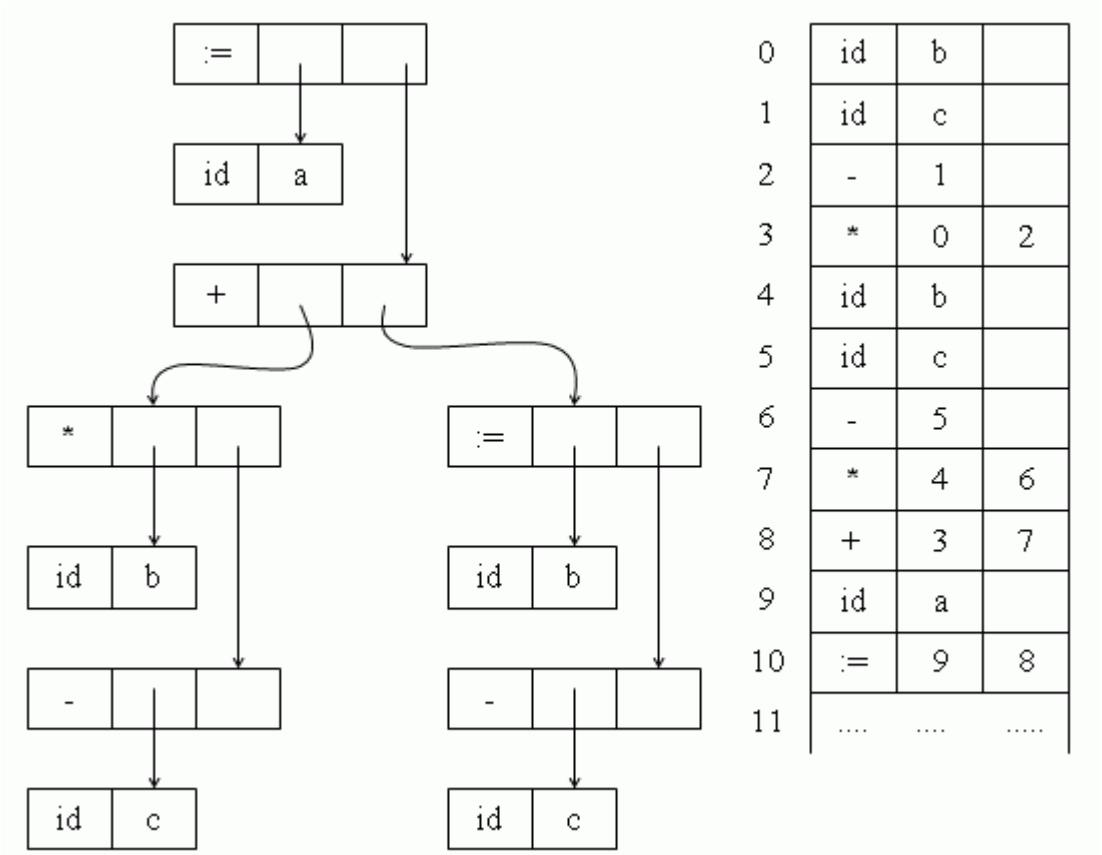
Ví dụ: a := b * - c + b * - c, ta có cây cú pháp và DAG:



Cây cú pháp có thể được cài đặt bằng một trong 2 phương pháp:

- Mỗi nút được biểu diễn bởi một mẫu tin, với một trường cho toán tử và các trường khác trỏ đến con của nó.
- Một mảng các mẩu tin, trong đó chỉ số của phần tử mảng đóng vai trò như là con trỏ của một nút.

Tất cả các nút trên cây cú pháp có thể tuân theo con trỏ, bắt đầu từ nút gốc tại 10



0	id	b	
1	id	c	
2	-	1	
3	*	0	2
4	id	b	
5	id	c	
6	-	5	
7	*	4	6
8	+	3	7
9	id	a	
10	:=	9	8
11	....	....	.....

2 KÍ PHÁP HẬU TỐ

Định nghĩa kí pháp hậu tố của một biểu thức:

1) E là một biến hoặc hằng số, kí pháp hậu tố của E là E.

2) E là biểu thức dạng: E_1 op E_2với op là toán tử 2 ngôi thì kí pháp hậu tố của E là: $E'_1E'_2$op với E'_1, E'_2 là kí pháp hậu tố của E_1, E_2 tương ứng.

3) Nếu E là biểu thức dạng (E_1), thì kí pháp hậu tố của E_1 cũng là kí pháp hậu tố của E.

Ví dụ: Kí pháp hậu tố của (9-5)+2 là 95-2+;

Kí pháp hậu tố của câu lệnh if a then if c-d then a+c else a*c else a+b là

a?(c-d?a+c:a*c):a+b tức là: acd-ac+ac*?ac+?

3 MÃ 3 ĐỊA CHỈ

Mã ba địa là một chuỗi các câu lệnh, thông thường có dạng: x:= y op z

X,y,z là tên, hằng do người lập trình tự đặt, op là một phép toán nào đó phép toán toán học, logic…

3.1 Một số câu lệnh ba địa chỉ thông dụng

1. Các câu lệnh gán có dạng *x := y op z*, trong đó *op* là một phép toán số học hai ngôi hoặc phép toán logic.

2. Các phép gán có dạng *x := op y*, trong đó *op* là phép toán một ngôi. Các phép toán một ngôi chủ yếu là phép trừ, phép phủ định logic, phép chuyển đổi kiểu, phép dịch bít.

3. Các câu lệnh sao chép dạng *x := y*, gán *y* vào *x*.

4. Lệnh nhảy không điều kiện *goto L*. Câu lệnh ba địa chỉ có nhãn *L* là câu lệnh được thực hiện tiếp theo.

5. Các lệnh nhảy có điều kiện như *if x relop y goto L*. Câu lệnh này thực hiện một phép toán quan hệ cho *x* và *y*, thực hiện câu lệnh có nhãn *L* nếu quan hệ này là đúng, nếu trái lại sẽ thực hiện câu lệnh tiếp theo.

6. Câu lệnh *param x* và *call p,n* dùng để gọi thủ tục. Còn lệnh *return y* để trả về một giá trị lưu trong *y*. Ví dụ để gọi thủ tục $p(x_1,x_2,...,x_n)$ thì sẽ sinh các câu lệnh ba địa chỉ tương ứng như sau:

param x_1

...

param x_n

call p, n

7. Các phép gán chỉ số có dạng x := y[i] có ý nghĩa là gán cho x giá trị tại vị trí i sau y.

tương tự đối với x[i] := y

8. Phép gán địa chỉ và con trỏ có dạng x := &y, x := *y, *x := y

3.2 Cài đặt các câu lệnh ba địa chỉ

3.2.1 Bộ tứ

Bộ tứ là cấu trúc bản ghi với bốn trường, được gọi là *op*, *arg1*, *arg2* và *result*.

Ví dụ: Câu lệnh a := -b * (c+d) được chuyển thành:

t1 := - b *t2 := c+d*

*t3 := t1 * t2* *a := t3*

Biểu diễn bằng bộ tứ như sau:

	Op	**arg1**	**arg2**	**result**
0	Uminus	b		t1
1	+	c	d	t2
2	*	t1	t2	t3
3	Assign	t3		a

3.2.2. Bộ ba

Để tránh phải đưa các tên tạm thời vào bảng ký hiệu, chúng ta có thể tham chiếu đến một giá trị tạm bằng vị trí của câu lệnh dùng để tính nó (tham chiếu đến câu lệnh đó chính là tham chiếu đến con trỏ chứa bộ ba của câu lệnh đó). Nếu

chúng ta làm như vậy, câu lệnh mã ba địa chỉ sẽ được biểu diễn bằng một cấu trúc chỉ gồm có ba trường op, arg1 và arg2.

Ví dụ trên sẽ được chuyển thành bộ ba như sau:

	op	**arg1**	**arg2**
0	uminus	b	
1	+	c	d
2	*	(0)	(1)
3	assign	a	(2)

Lệnh sao chép đặt kết quả trong arg1 và tham số trong arg2, toán tử là assign.

Các số trong ngoặc tròn biểu diễn các con trỏ chỉ đến một cấu trúc bộ ba, còn con trỏ chỉ đến bảng ký hiệu được biểu diễn bằng chính các tên. Trong thực hành, thông tin cần để diễn giải các loại mục ghi khác nhau trong *arg1* và *arg2* có thể được mã hoá vào trường op hoặc đưa thêm một số trường khác.

Phép toán ba ngôi như x[i] := y cần đến hai mục trong cấu trúc bộ ba:

	op	**arg1**	**arg2**
(0)	[]=	x	i
(1)	assign	(0)	y

Phép toán ba ngôi x := y[i]

Tc t	**op**	**arg1**	**arg2**
(0)	[]=	y	i
(1)	assign	x	(0)

3.3 Cú pháp điều khiển sinh mã ba địa chỉ

Đối với mỗi ký hiệu X, ký hiệu:

- X.place là nơi để chứa mã ba địa chỉ sinh ra bởi X (dùng để chứa các kết quả trung gian). Vì thế sẽ có một hàm định nghĩa là newtemp dùng để sinh ra một biến trung gian (biến tạm) để gán cho X.place.
- X.code chứa đoạn mã ba địa chỉ của X
- Thủ tục gen để sinh ra câu lệnh ba địa chỉ.

3.3.1 Sinh mã ba địa chỉ cho biểu thức số học

Sản xuất	**Luật ngữ nghĩa**
S -> id := E	S.code := E.code || gen(id.place ‘:=’ E.place)

E -> E_1 + E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
E -> E_1 * E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
E -> - E_1	E.place := newtemp; E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)
E -> (E_1)	E.place := E_1.place E.code := E_1.code
E -> id	E.place := id.place E.code := ''

Thuộc tính S.code biểu diễn mã 3 địa chỉ cho lệnh gán S. Ký hiệu E có 2 thuộc tính E.place là giá trị của E và E.code là chuỗi lệnh 3 địa chỉ để đánh giá E.

Khi mã lệnh 3 địa chỉ được sinh, tên tạm được tạo ra cho mỗi nút trong trên cây cú pháp.

Giá trị của ký hiệu chưa kết thúc E trong luật sinh E → E_1 + E_2 được tính vào trong tên tạm t. Nói chung mã lệnh 3 địa chỉ cho lệnh gán id := E bao gồm mã lệnh cho việc đánh giá E vào trong biến tạm t, sau đó là một lệnh gán *id.place := t*.

Hàm newtemp trả về một chuỗi các tên t_1, t_2, ... , t_n tương ứng các lời gọi liên tiếp. *Gen (x ':=' y '+' z)* để biểu diễn lệnh 3 địa chỉ *x :=y+z*

Ví dụ:

Hãy sinh mã ba địa chỉ cho câu lệnh sau "x := a + (b * c)"

S

=> x := E

=> x := E_1 + E_2

=> x := a + E_2

=> x := a + (E_3)

=> x := a + (E_4 * E_5)

=> x := a+ (b * E_5)

=> x := a + (b * c)

E_5.place := c E_5.code := ''

E_4.place := b E_4.code := ''

E_3.place := t_1 E_3.code := t_1 := b * c

E_2.place := t_1 E_2.code := t_1 := b * c

E_1.place := a E_1.code := ‘’

E_1.place := a E_1.code := ‘’

E.place := t_2 E.code := t_1 := b * c || t_2 := a + t_1

S.code := t_1 := b * c || t_2 := a + t_1 || x := t_2

3.3.2 Sinh mã ba địa chỉ 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.

Ví dụ: E có dạng *a<b*. Thế thì mã sinh ra sẽ có dạng

if a<b goto E.true

goto E.false

Ví dụ đoạn lệnh sau: ***if a>b then*** *a:=a-b;*

else *b:=b-a;*

được chuyển thành mã ba địa chỉ như sau *E.true = L1 và E.false = L2*

if a>b goto L1

goto L2

L1: t1 := a –b

a := t1

goto Lnext

L2: t2 := b-a

b := t2

Lnext:

Một số cú pháp điều khiển sinh mã ba địa chỉ cho các biểu thức Boole.

Sản xuất	**Luật ngữ nghĩa**
E -> E_1 or E_2	E_1.true := E.true; E_1.false := newlable; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ‘:’) || E_2.code

E -> E_1 and E_2	E_1.true := newlable; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ':') || E_2.code
E -> not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code;
E -> (E_1)	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code;
E -> id_1 relop id_2	E.code := gen('if' id_1.place relop.op id_2.place 'goto' E.true) || gen('goto' E.false)
E -> true	E.code := gen('goto' E.true)
E -> false	E.code := gen('goto' E.false)

Ví dụ: Sinh mã ba địa chỉ cho đoạn chương trình sau:

if a>b and c>d then x:=y+z

else x:=y-z

Lời giải:

Nếu coi E là biểu thức logic *a>b and c>d* thì đoạn chương trình trên trở thành *if E then x:=y+z* , khi đó mã ba địa chỉ cho đoạn chương trình có dạng:

E.code {
if E=true goto E.true
goto E.false }
E.true: t1:= y+z
x := t1;
E.false :
t2 := y-z
x :=t2

Áp dụng các luật sinh mã ba địa chỉ trong bảng trên chúng ta có đoạn mã ba địa chỉ cho đoạn chương trình nguồn ở trên là:

if a>b goto L1
goto L3

L1: if c>d goto L2

goto L3

L2: t1 := y+z

x := t1

goto L4

L3: t2 := y-z

x := t2

L4:

3.3.3 Biểu thức logic và biểu thức số học

Xét văn phạm E → E+ E | E and E | E relop E | id

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

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

Ta có luật ngữ nghĩa kết hợp với E → E1 + E2 như sau

E.type := arith;

if *E_1.type = arith* **and** *E_2.type = arith* **then begin**

/ phép cộng số học bình thường */*

E.place := newtemp;

E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)

end

else if *E_1.type = arith* **and** *E_2.type = bool* **then begin**

E.place := newtemp;

E_2.true := newlabel;

E_2.false := newlabel;

E.code := E_1.code || E_2.code || gen(E_2.true ':' E.place ':= ' E_1.place +1) || gen('goto' nextstat +1) || gen(E_2.false ':' E.place ':= ' E_1.place)

else if ...

Trong trường hợp nếu có biểu thức logic nào có biểu thức số học, chúng ta sinh mã lệnh cho E1, E2 bởi các lệnh

E_2.true : E.place := E_1.place +1

goto nextstat +1

E_2.false : E.place := E_1.place

3.3.4 Sinh mã ba địa chỉ cho một số lệnh điều khiển

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

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.

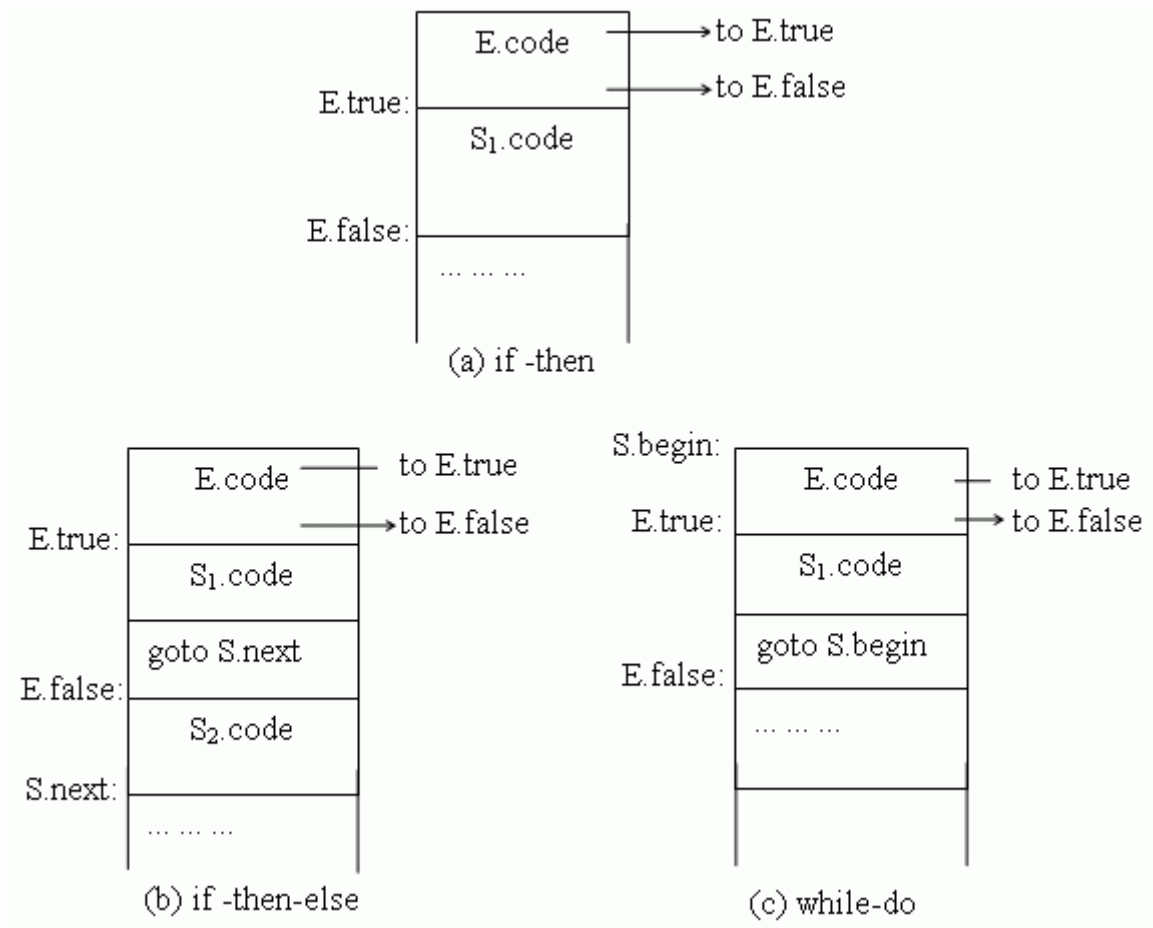
***** *Dịch biểu thức logic trong các lệnh điều khiển*

- 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.



Định nghĩa trực tiếp cú pháp cho việc dịch các biểu thức logic thành mã lệnh 3 địa chỉ. Chú ý true và false là các thuộc tính kế thừa.

Sản xuất	**Luật ngữ nghĩa**
S -> if E then S_1	E.true := newlable; E.false := S.next; S_1.next := S.next; S.code := E.code || gen(E.true ':') || S_1.code
S -> if E then S_1 else S_2	E.true := newlable; E.false := newlable; S_1.next := S.next; S_2.next := S.next; S.code := E.code || gen(E.true ':') || S_1.code || gen('goto' S.next) || gen(E.false ':') || S_2.code
S -> while E do S_1	S.begin := newlable; E.true := newlable;

	E.false := S.next S_1.next := S.begin; S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)

Ví dụ 1: Cho mã nguồn sau:

while a<>b do
if a>b then a:=a-b
else b:=b-a

Mã ba địa chỉ:

L1: if a<>b goto L2
goto Lnext
L2: if a>b goto L3
goto L4
L3: t1 := a-b
a := t1
goto L1
L4: t2 := b-a
b := t2
goto L1
Lnext:

3.3.5.Các khai báo.

Đối với các khai báo định danh, ta không sinh ra mã lệnh tương ứng trong mã ba địa chỉ mà dùng bảng ký hiệu để lưu trữ. *Như vậy có thể hiểu là kết quả của sinh mã ba địa chỉ từ chương trình nguồn là tập lệnh ba địa chỉ và bảng ký hiệu quản lý các định danh.*

Với mỗi định danh, ta lưu các thông tin về kiểu và địa chỉ tương đối để lưu giá trị cho định danh đó.

Ví dụ:

Giả sử ký hiệu offset để chứa địa chỉ tương đối của các định danh; mỗi số interger chiếm 4 byte, số real chứa 8 byte và mỗi con trỏ chứa 4 byte; giả sử hàm enter dùng để nhập thông tin về kiểu và địa chỉ tương đối cho một định danh, chúng ta có ví dụ dưới đây mô ta việc sinh thông tin vào bảng ký hiệu cho các khai báo.

Sản xuất	**Luật ngữ nghĩa**

P -> D	offset := 0
D -> D ; D	
D -> id : T	enter(id.name,T.type, offset) ; offset := offset + T. width
T -> interger	T.type := interger; T. width := 4
T -> real	T.type := real; T. width := 8
T -> array [num] of T_1	T.type := array(num.val,T_1.type); T.width := num.val * T_1. width
T -> ^T_1	T.type := pointer(T_1.type) T. width := 4

Trong các đoạn mã ba địa chỉ, khi đề cập đến một tên, ta sẽ tham chiếu đến bảng ký hiệu để lấy thông tin về kiểu, địa chỉ tương ứng để sử dụng trong các câu lệnh. *Chú ý:* Địa chỉ tương đối của một phần tử trong mảng, ví dụ x[i], được tính bằng địa chỉ của x cộng với i lần độ dài của mỗi phần tử

BÀI TẬP

Câu 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;}
}
```

Câu 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.

Câu 3: 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.

Câu 4:

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)

Câu 5:

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]

Câu 6. Dịch lệnh gán sau thành mã máy 3 - địa chỉ :

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