CHƯƠNG 5 BIÊN DỊCH DỰA CÚ PHÁP.

1. MỤC ĐÍCH, NHIỆM VỤ.

- Các hành động dịch phụ thuộc rất nhiều vào cú pháp của chương trình nguồn cần dịch.Quá trình dịch được điều khiển theo cấu trúc cú pháp của chương trình nguồn, cú pháp này được xác định thông qua bộ phân tích cú pháp.
- Nhằm điều khiển các phần hoạt động theo cú pháp, cách thường dùng là gia cố các luật sản xuất (mà ta biết cụ thể những luật nào và thứ tự thực hiện ra sao thông qua cây phân tích) bằng cách thêm các thuộc tính cho văn phạm đấy, và các qui tắc sinh thuộc tính gắn với từng luật cú pháp. Các qui tắc đó, ta gọi là qui tắc ngữ nghĩa (semantic rules).
- thực hiện các qui tắc ngữ nghĩa đó sẽ cho thông tin về ngữ nghĩa, dùng để kiểm tra kiểu, lưu thông tin vào bảng ký hiệu và sinh mã trung gian.
- Có hai tiếp cận để liên kết (đặc tả) các qui tắc ngữ nghĩa vào các luật cú pháp (sản xuất) là cú pháp điều khiển (syntax-directed definition) và lược đồ dịch (translation scheme).
- Các luật ngữ nghĩa còn có các hành động phụ (ngoài việc sinh thuộc tính cho các ký hiệu văn phạm trong sản xuất) như in ra một giá trị hoặc cập nhật một biến toàn cục.

Các kiến thức trong phần này không nằm trong khối chức năng riêng rẽ nào của chương trình dịch mà được dùng làm cơ sở cho toàn bộ các khối nằm sau khối phân tích cú pháp.

Một xâu vào →Cây phân tích →Đồ thị phụ thuộc →thứ tựđánh giá cho các luật ngữ nghĩa.

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, được chia thành 2 tập con là thuộc tính tổng hợp (synthesized attribute) và thuộc tính kế thừa (inherited attribute) của ký hiệu văn phạm đó.

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).

2.1. 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->α 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.

<u>Ví dụ</u>: 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.

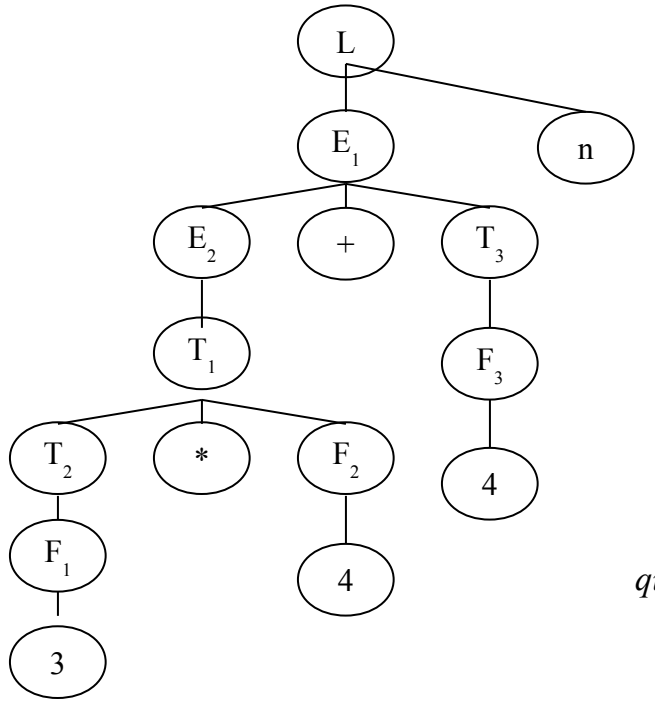
2.1.2. *Thuộc tính tổng hợp.*

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



<u>*ví dụ 1*</u>

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.

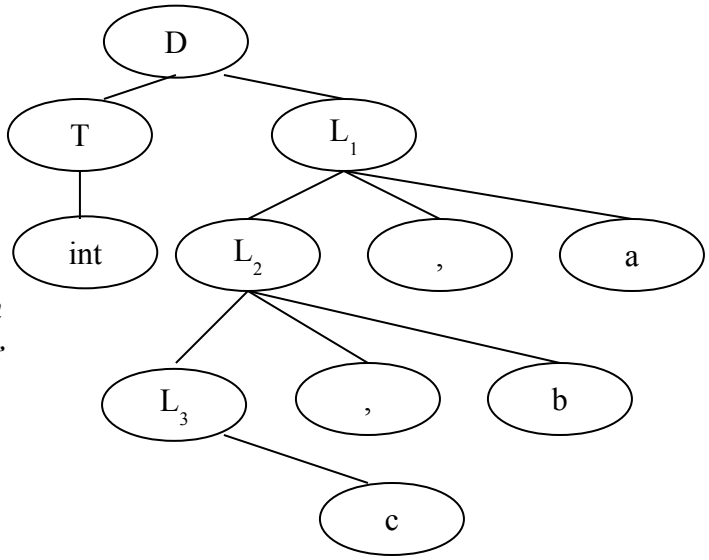
Thuộc tính kế thừa (inherited attribute) là thuộc tính tại một nút có giá trị được xác định theo giá trị thuộc tính của cha hoặc anh em của nó.

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:

sản xuất	luật ngữ nghĩa
D -> T L	L.in := T.type
T -> int	T.type := interger
T -> real	T.type := real
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))

Bài luyện tập:

1) Cho văn phạm sau định nghĩa một số ở hệ cơ số 2

B -> 0 | 1 | B 0 | B 1

Hãy định nghĩa một cú pháp điều khiển để dịch một số ở hệ cơ số 2 thành một số ở hệ cơ số 10 (hay nói cách khác là tính giá trị của một số ở hệ cơ số 2). Xây dựng cây đánh dấu(xây dựng cây cú pháp cùng với giá trị thuộc tính trên mỗi nút) với đầu vào là "1001".

Mở rộng: sinh viên tự làm bài toán này với các sản xuất định nghĩa một số thực ở hệ cơ số 2:

S->L.L | L
L->LB | B
B->0 | 1

<u>Lời giải:</u> Định nghĩa thuộc tính tổng hợp val của ký hiệu B để chứa giá trị tính được của số biểu diễn bởi B.

xuất phát từ cách tính:

$(a_n a_{n-1} \ldots a_1 a_0)_2 \quad := a_n*2^n+a_{n-1}*2^{n-1}+\ldots+a_1*2+a_0$
$:= 2*(a_n*2^{n-1}+\ldots+a_1)+a_0$
$:= 2*(a_n\ldots a_1)+a_0$

Do đó nếu có

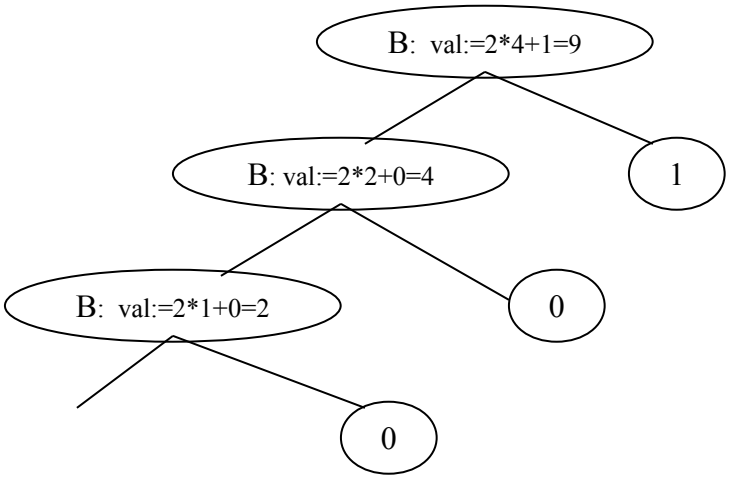
*B -> B_1 1 thì B.val := 2*B_1.val+1*

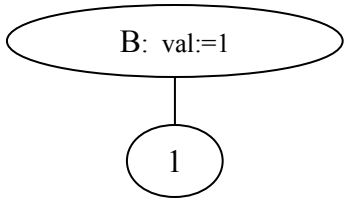
*B -> B_1 0 thì B.val := 2*B_1.val*

Vì vậy, chúng ta xây dựng các luật dịch như sau:

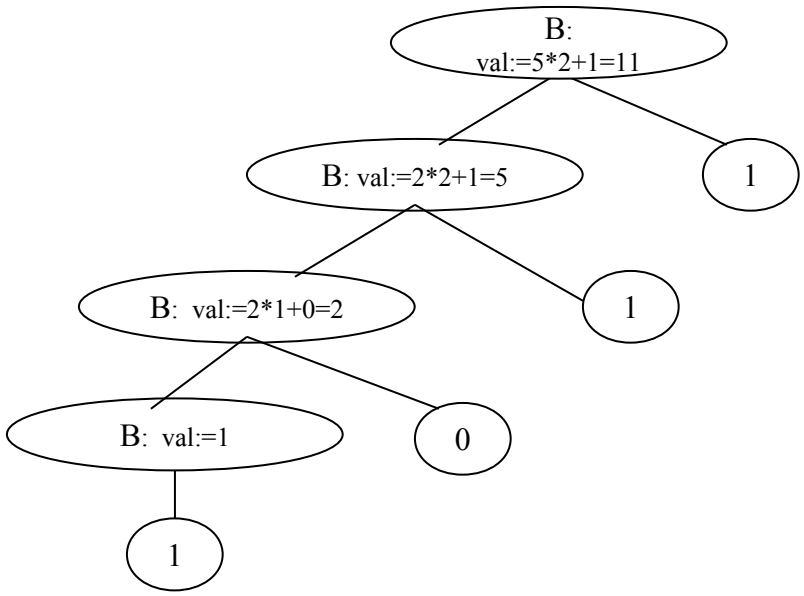
Luật phi ngữ cảnh	***Luật dịch***
B->0	*B.val=0;*
B->1	*B.val:=1;*
B->B_1 0	*B.val:=2*B_1.val +0*
B->B 1	*B.val:=2*B_1.val+1*

Cây đánh dấu:





Xét một cây đánh dấu khác cho xâu vào "1011"



2.2. 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.

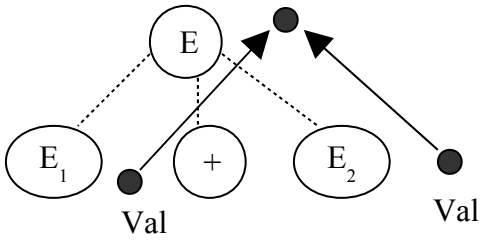
Trước khi xây dựng một đồ thị phụ thuộc cho một cây phân tích cú pháp, chúng ta chuyển mỗi hành động ngữ nghĩa thành dạng b := f(c_1,c_2,. . .,c_k) bằng cách dùng một thuộc tính tổng hợp giả b cho mỗi hành động ngữ nghĩa có chứa một lời gọi thủ tục. Đồ thị này có một nút cho mỗi thuộc tính, một cạnh 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*

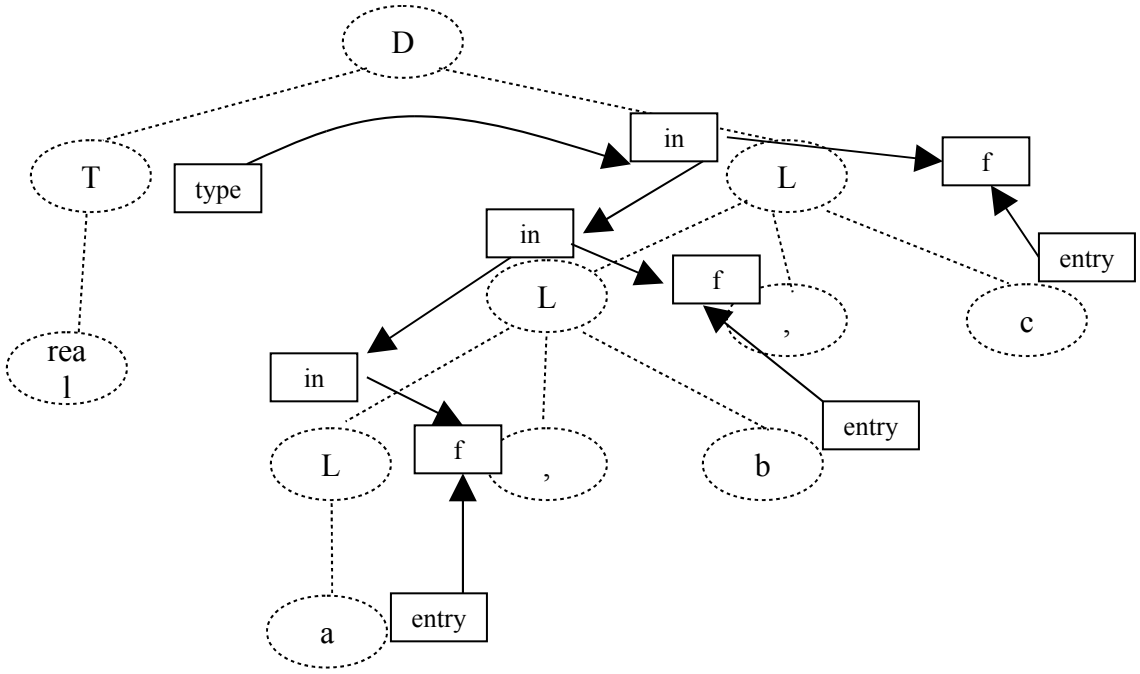
VD 1: Dựa vào cây phân tích (nét đứt đoạn) và luật ngữ nghĩa ứng với sản xuất ở bảng, ta thêm các nút và cạnh thành đồ thị phụ thuộc:

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 -> real	T.type := real
L -> L_1, id	L_1.in := L.in ; addtype(id.entry, L.in)
L -> id	addtype(id.entry,L.in)



2.3. Thứ tự đánh giá thuộc tính.

Trên đồ thị DAG được xây dựng như ví dụ trên, chúng 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 $m_1, m_2, \ldots, m_k$ thì nếu có $m_i \rightarrow m_j$ 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 $c_1, c_2, \ldots, c_k$ trong một hành động ngữ nghĩa $b:=f(c_1, c_2, \ldots, c_k)$ đã được tính trước khi ta ước lượng f.
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. Tức là xây dựng cây cú pháp với mỗi nút (đỉnh) đại diện cho một ký hiệu văn phạm sau đó mới xây dựng đồ thị phụ thuộc theo thuật toán 5.1

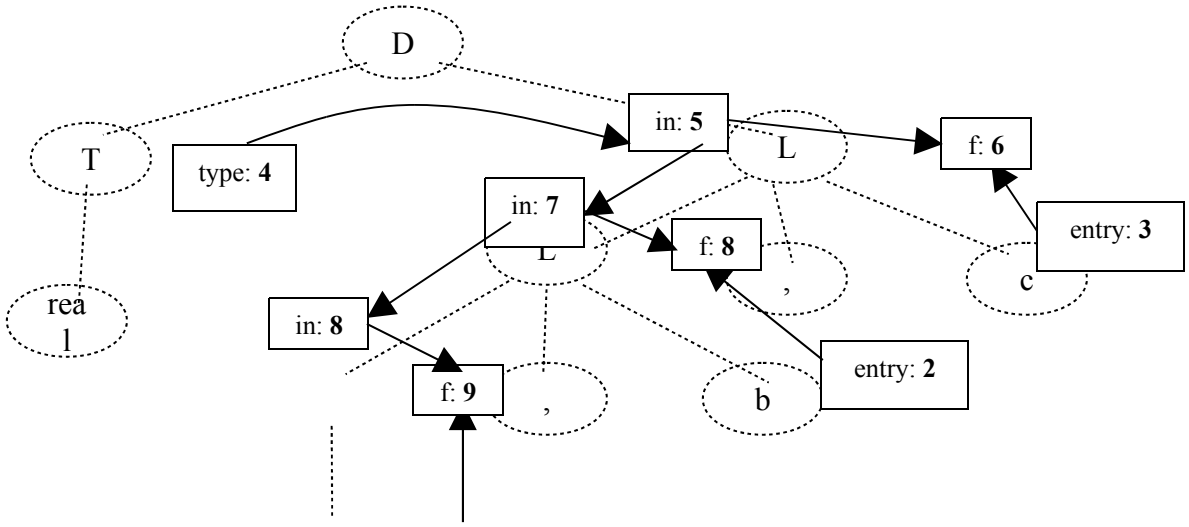
+ trong đồ thị phụ thuộc, mỗi nút đại diện cho một thuộc tính của một ký hiệu văn phạm.

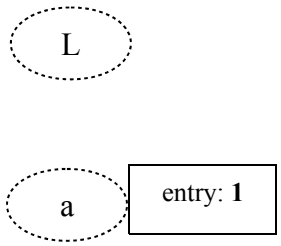
+ có thể một loại thuộc tính này lại phụ thuộc vào một loại thuộc tính khác, chứ không nhất thiết là chỉ các thuộc tính cùng loại mới phụ thuộc vào nhau. Trong ví dụ trên, thuộc tính *entry* phụ thuộc vào thuộc tính *in*.

+ 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.

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 "real 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

1. *Phương pháp dùng cây phân tích cú pháp*. Kết quả trả về của phân tích cú pháp phải là cây phân tích cú pháp, sau đó xây dựng một thứ tự duyệt hay một sắp xếp topo của đồ thị từ cây phân tích cú pháp đó. Phương pháp này không thực hiện được nếu đồ thị phụ thuộc có "vòng".
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.
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.

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.
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 thứ tự duyệt tự nhiên đặc trưng cho nhiều phương pháp dịch Top-down và Bottom-up là thủ tục duyệt theo chiều sâu (depth-first order). Thủ tục duyệt theo chiều sâu được trình bày như dưới đây:

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
```

Một lớp các cú pháp điều khiển được gọi là cú pháp điều khiển thuần tính L hay gọi là điều khiển thuần tính L (L-attributed definition) có các thuộc tính luôn có thể tính toán theo chiều sâu.

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

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.

Ta thấy nếu ngôn ngữ mà ngữ nghĩa của một từ tố được xác định chỉ phụ thuộc vào ngữ cảnh bên trái (các từ tố bên trái) thì một phương pháp duyệt cú pháp từ trái sang phải cho đầu vào có thể kết hợp với điều khiển ngữ nghĩa để duyệt cả cú pháp và ngữ nghĩa đồng thời. Từ đó, ta thấy cú pháp điều khiển thuần tính L thoả mãn điều kiện này. Hay với cú pháp điều khiển thuần tính L, ta có thể duyệt đầu vào từ trái sang phải để sinh các thông tin cú pháp và ngữ nghĩa một cách đồng thời. Với phương pháp phân tích cú pháp tuyến tính LL và LR, ta có thể kết hợp để thực hiện cả các hành động ngữ nghĩa thuần tính L.

Nhưng 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.

Thực hiện hành động ngữ nghĩa trong phân tích LL

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 một ký hiệu không kết thúc được gắn với một hàm thực hiện. Giả sử với ký hiệu không kết thúc 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-> $\alpha_1 | \alpha_2 | ... | \alpha_n$

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

```
void ParseA(Symbol A, Rule r, ...)
{       if(r==A->α 1)
                gọi hàm xử lý ngữ nghĩa tương ứng luật A->α 1
        else if(r==A->α 2)
                gọi hàm xử lý ngữ nghĩa tương ứng luật A->α 2
        . . .
        else if(r==A->α n)
                gọi hàm xử lý ngữ nghĩa tương ứng luật A->α n
}
```

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 -> $X_1 ... X_k$ với $\alpha_i = X_1 ... X_k$

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 ->    T {R.i:=T.val}
        R {E.val:=R.s}
R ->    +
        T {R1.i:=R.i+T.val}
        R1 {R.s:=R1.s}
R ->    ε   {R.s:=R.i}
T ->    ( E ) {T.val:=E.val}
T ->    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()***
Bây giờ ta 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 $		
$R1T	num2 $	T->num2	T.val=4
$R1num2	num2 $		
$R1	$	R1->ε	R1.i=T.val=4
$	$		R1.s=T.val+R1.i=10
			R.s=R1.s=10
			E.val=R.s=10

Nhận xét:

Mọi cú pháp điều khiển thuần tính L dựa trên văn phạm LL(1) đều có thể kết hợp quá trình phân tích cú pháp tuyến tính với việc thực hiện các hành động ngữ nghĩa.

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, đối với 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

Chẳng hạn chúng ta sẽ 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, chúng ta lại đặt vào trong ngăn xếp giá trị 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 chúng ta cũng loại bỏ giá trị tương ứng với nó ra khỏi ngăn xếp giá trị. Chúng ta có thể xem qua quá trình phân tích gạt, thu gọn với ví dụ cho xâu vào "3*5+4":

chú ý:

+ phân tích từ tố cho ta kết quả xâu vào là (ký hiệu d là digit):

d1(3)*d2(5)+d3(4)n

+ với ký hiệu không có giá trị *val*, chúng 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->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->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.

Sau đây, chúng ta giới thiệu một số cú pháp điều khiển khác mà cũng có thể thực hiện khi phân tích LR bằng một số kỹ thuật:

1) **loại bỏ việc gắn kết các hành động ngữ nghĩa ra khỏi lược đồ dịch**
2) **kế thừa các thuộc tính trên ngăn xếp**
3) **Mô phỏng thao tác đánh giá các thuộc tính kế thừa**
4) **Thay thuộc tính kế thừa bằng thuộc tính tổng hợp**

Sinh viên tự tham khảo trong tài liệu các phần này.

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

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

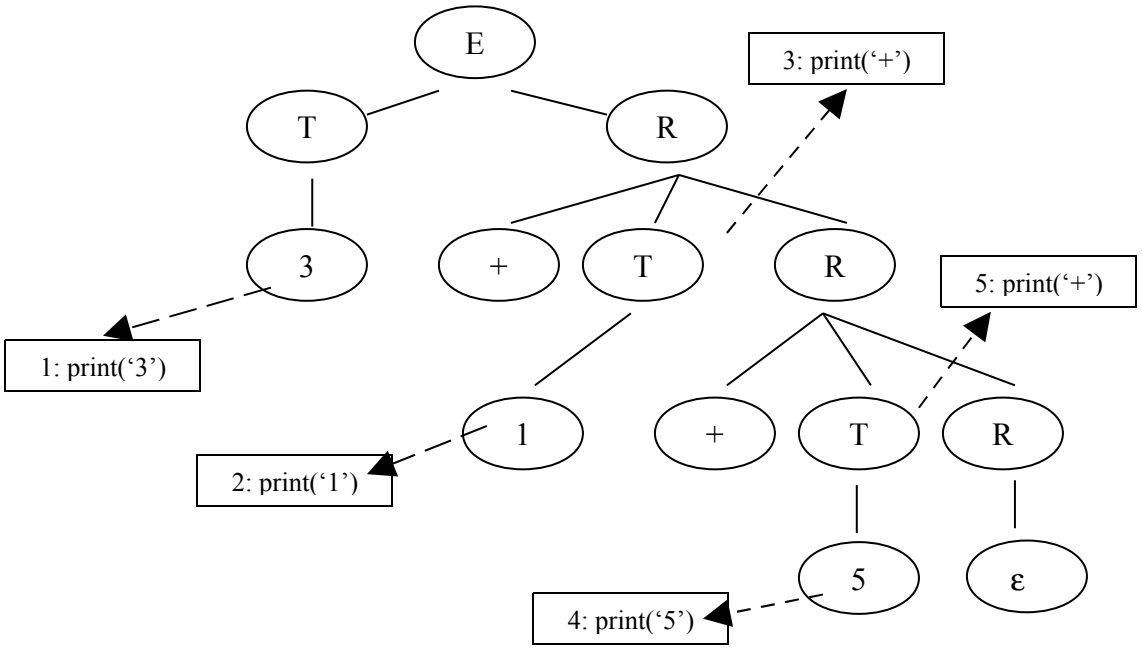
+ Lược đồ dịch xác định thứ tự thực hiện hành động ngữ nghĩa trong mỗi sản xuất

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"

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



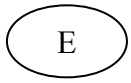
Kết quả dịch là "3 1 + 5 +"

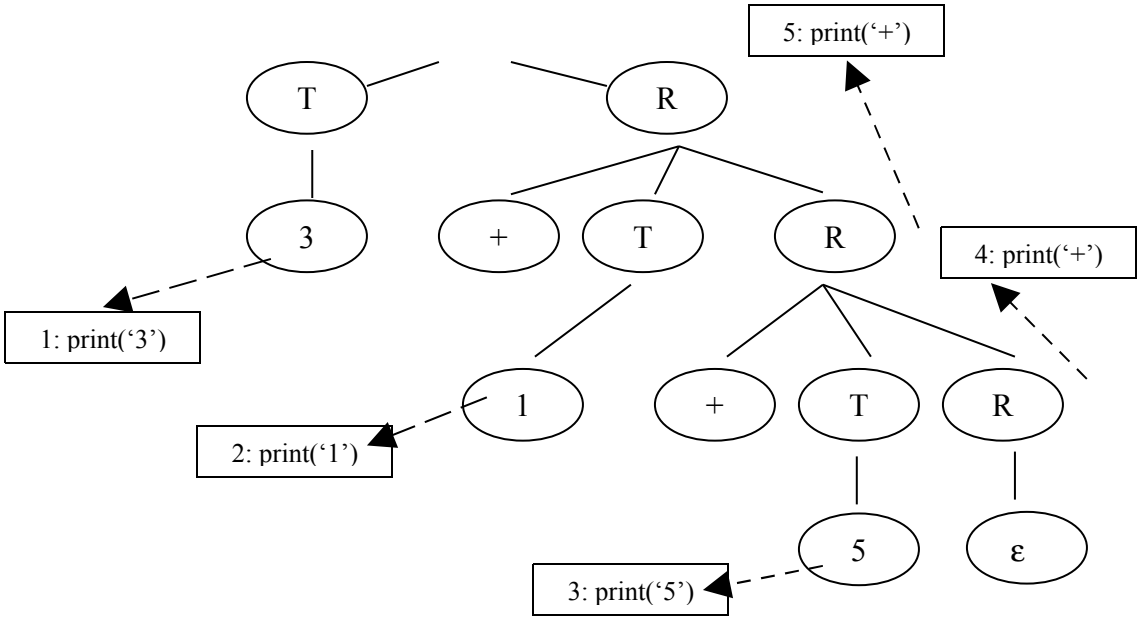
Chú ý là nếu trong lược đồ dịch ta đặt hành động ngữ nghĩa ở vị trí khác đi, chúng ta sẽ có kết quả dịch khác ngay. Ví dụ, đối với lược đồ dịch, ta thay đổi một chút thành lược đồ dịch như sau:

E -> T R
R -> + T R *{print('+')}*
R -> ε
T -> num *{print(num.val)}*

Xét biểu thức "3+1+5"

Chúng ta 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:

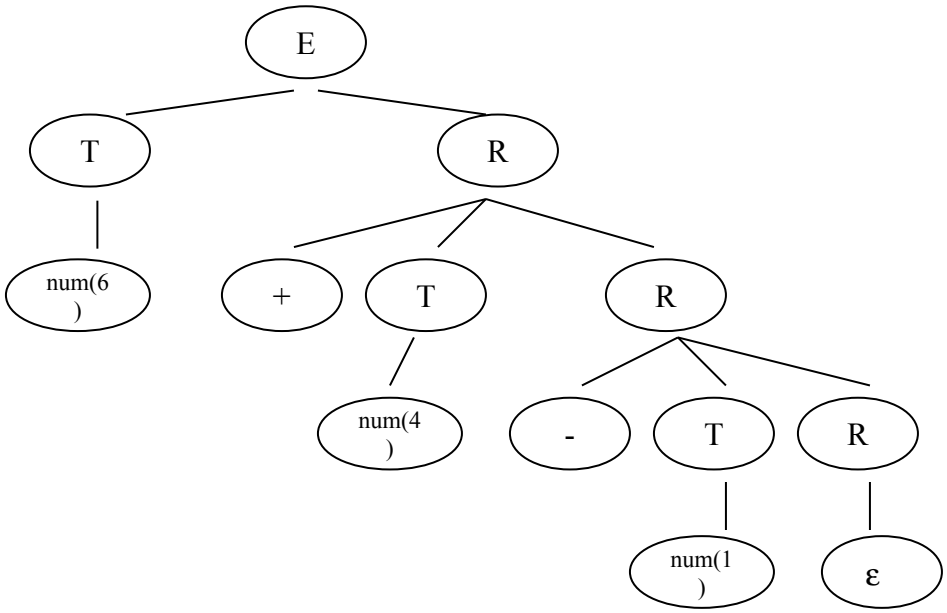
Cho văn phạm biểu diễn biểu thức gồm các toán tử + và - với toán hạng là các số:

E -> T R
R -> + T R
R -> - T R
R -> ε
T -> (E)
T -> num

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

<u>Giải đáp:</u>

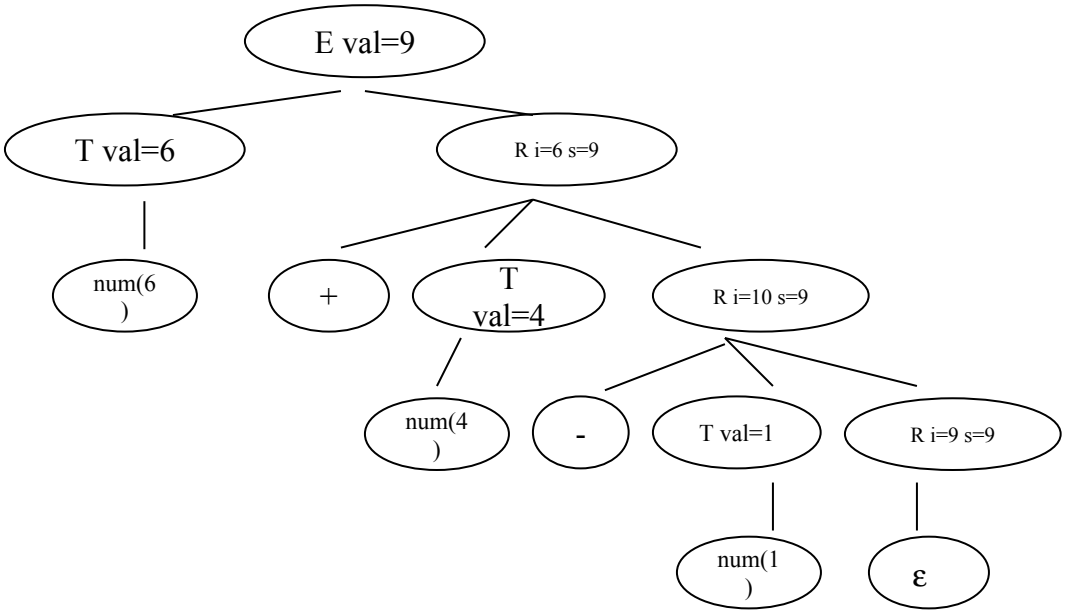
Trước hết, chúng ta thử xem cây phân tích cú pháp cho đầu vào "6+4-1"



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. Chúng ta đặt R.i chứa giá trị của phần biểu thức đứng trước R và R.s chứa kết quả. Chúng ta xây dựng lược đồ dịch như sau:

E -> T $\{R.i:=T.val\}$
R $\{E.val:=R.s\}$
R -> +
T $\{R_1.i:=R.i+T.val\}$
R_1 $\{R.s:=R_1.s\}$
R -> -
T $\{R_1.i:=R.i-T.val\}$

R_1 *{R.s:=R_1.s}*
R -> ε *{R.s:=R.i}*
T -> (E) *{T.val:=E.val}*
T -> num *{T.val:=num.val}*



<u>Lưu ý:</u>

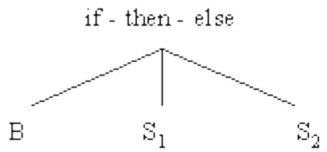
nếu chúng ta xác định một cách duyệt khác cách duyệt theo hướng sâu thì cách đặt hành động dịch vào vị trí nào sẽ được làm khác đi. Tuy nhiên cách duyệt theo hướng sâu là cách duyệt phổ biến nhất và tự nhiên nhất (vì ngữ nghĩa sẽ được xác định dần theo chiều duyệt đầu vào từ trái sang phải) nên chúng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

4. DỰNG CÂY CÚ PHÁP.

4.1. Cây cú pháp.

Cây cú pháp (syntax - tree) là dạng rút gọn của cây phân tích cú pháp dùng để biểu diễn cấu trúc ngôn ngữ. Trong cây cú pháp các toán tử và từ khóa không phải là nút lá mà là các nút trong.

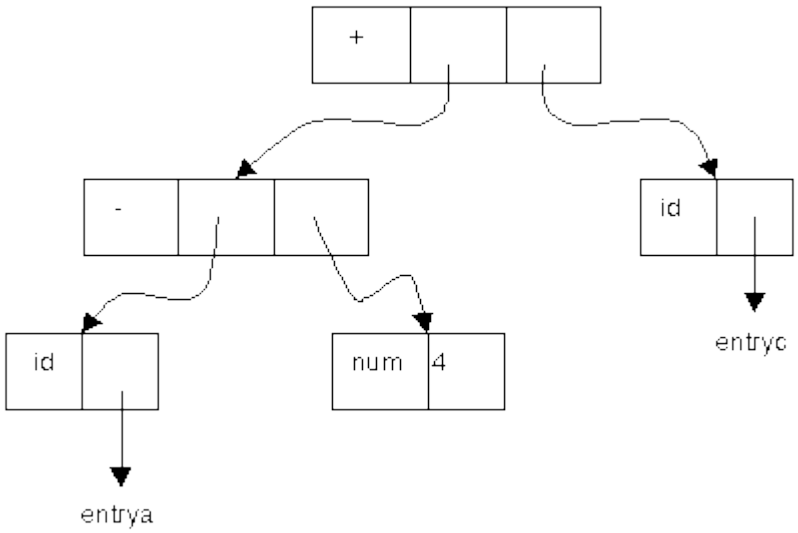
Ví dụ với luật sinh S → if B then S1 else S2 được biểu diễn bởi cây cú pháp:



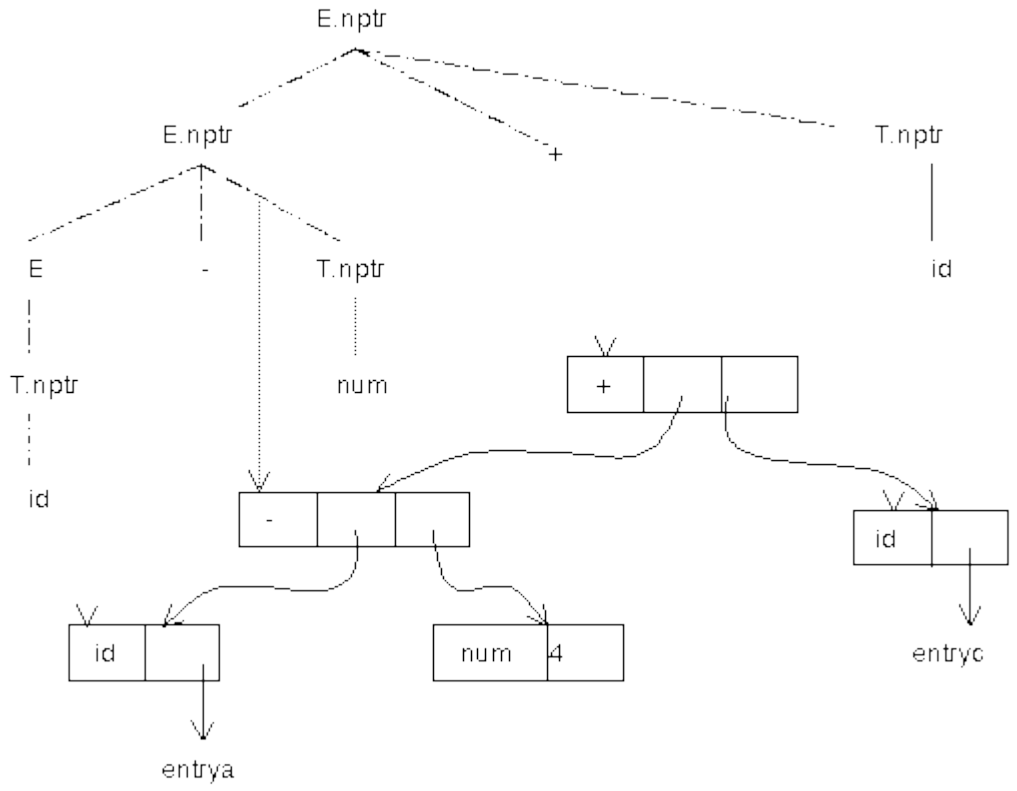
- Một sự đơn giản khác của cây cú pháp là chuỗi các luật sinh đơn được rút gọn lại. Chẳng hạn ta có:

Xây dựng cây cú pháp cho biểu thức.

Tương tự như việc dịch một biểu thức thành dạng hậu tố. Xây dựng cây con cho biểu thức con bằng cách tạo ra một nút cho toán hạng và toán tử. Con của nút toán tử là gốc của cây con biểu diễn cho biểu thức con toán hạng của toán tử đó. Mỗi một nút có thể cài đặt bằng một 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 ó nhãn là op và hai trờng chứa con trỏ, trỏ tới left và right.
2/ mkleaf(id, entry): Tạo một nút lá với nhãn là id và một trờng chứa con trỏ entry, trỏ tới ô trong bảng ký hiệu danh biểu.
3/ mkleaf(num,val): Tạo một nút lá với nhãn là num và trờng 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) (4): p4 := mkleaf(id, entryc)
(2): p2 := mkleaf(num,4) (5): p5 := mknode(" +", p3, p4)
(3): p3 := mknode(" -", p1, p2)
Cây được xây dựng từ dưới lên
entrya là con trỏ, trỏ tới ô của a trong bảng ký hiệu
entryc là con trỏ, trỏ tới ô của c trong bảng ký hiệu



* 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 giúp việc xây dựng cây cú pháp cho biểu thức là:
Luật sinh Luật ngữ nghĩa
E →E1 + T E.nptr := mknode('+', E1.nptr, T.nptr)
E →E1 - T E.nptr := mknode('-', E1.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)
Với biểu thức a - 4 + c ta có cây phân tích cú pháp (biểu diễn bởi đường chấm)

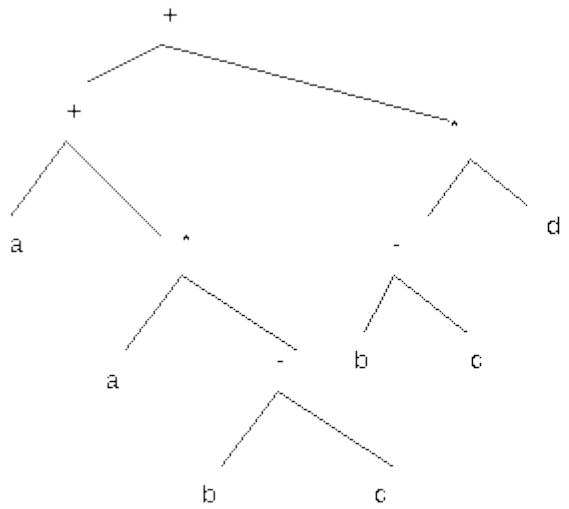


Luật ngữ nghĩa cho phép tạo ra cây cú pháp. Cây cú pháp có ý nghĩa về mặt cài đặt còn cây phân tích cú pháp chỉ có ý nghĩa về mặt logic.

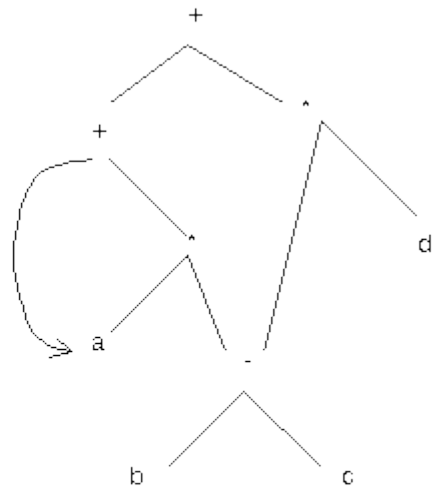
4.3. Đồ thị DRAG.

DAG (Directed Acyclic Graph): Đồ thị bao gồm các đỉnh chứa các thuộc tính và cỏc cạnh cú hướng để biểu thị sự phụ thuộc giữa các đỉnh. Cũng giống như cây cú pháp, tuy nhiên trong cây cú pháp các biểu thức con giống nhau 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



Cây cú pháp



DAG

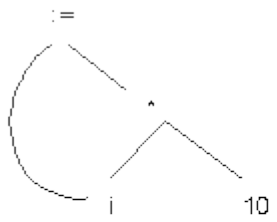
Để xây dựng một DAG, trước khi tạo một nút phải kiểm tra xem nút đó đã tồn tại chưa, nếu đã tồn tại thì hàm tạo nút (mknode, mkleaf) trả về con trỏ của nút đã tồn tại, nếu chưa thì tạo nút mới.

<u>Cài đặt DAG</u>

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.

Ví dụ:

Lệnh gán DAG Biểu diễn i := i + 10



id	entry i	
num	10	
+	1	2
:=	1	3

Nút 1: có nhãn là id, con trỏ trỏ tới entry i.
Nút 2: có nhãn là num, giá trị là 10.
Nút 3: có nhãn là +, con trái là nút 1, con phải là nút 2.
Nút 4: có nhãn là :=, con trái là nút 1, con phải là nút 3.

Giải thuật 5.1: Phương pháp value_number để xây dựng một nút trong DAG.

Giả sử rằng các nút được lưu trong một mảng và mỗi nút đợc tham khảo bởi số giá trị của nó. Mỗi một nút toán tử là một bộ ba <op, l, r >

Input: Nhãn op, nút l và nút r.

Output: Một nút với <op, l, r>

Phương pháp: Tìm trong mảng một nút m có nhãn là op con trái là l, con phải là r. Nếu tìm thấy thì trả về m, ngợc lại tạo ra một nút mới n, có nhãn là op, con trái là l, con phải là r và trả về m.

CHƯƠNG 6 PHÂN TÍCH NGỮ NGHĨA.

1. MỤC ĐÍCH NHIỆM VỤ.

Nhiệm vụ: 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.).*

Xét 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.*

Trong phạm vi tài liệu này, ta chỉ xét một số dạng trong kiểm tra kiểu của chương trình nguồn.

2. 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). <u>Kiểu hợp thành:</u>

+ *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: 0 interger x interger -> pointer(interger)

3. 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. Các vấn đề được nghiên cứu trong phần cú pháp điều khiển và lược đồ dịch.

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 để tiếp tục kiểm tra. trước hết nó thông báo về lỗi mô tả và vị trí lỗi. Lỗi xấut hiện gây ảnh hưởng đếncác 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.

3.1. Một số luật ngữ nghĩa kiểm tra kiểu

Đối với câu lệnh không có giá trị, ta có thể gán cho nó kiểu cơ sở đặc biệt void. Nếu có lỗi về kiểu được phát hiện trong câu lệnh 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:

VD1: Văn phạm cho khai báo:

D -> D ; D
D -> id : T
T -> interger
T -> char
T -> ^ T
T -> array [num] of T

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)

VD2: Văn phạm sau cho biểu thức

S -> id := E
E -> E + E
E -> E mod E

E -> E_1 [E_2]

E -> num

E -> id

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

VD3: Kiểm tra kiểu cho các câu lệnh:

S -> if E then S

S -> while E do S

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

VD4: Kiểu hàm: luật cú pháp sau đây thể hiện lời gọi hàm: E -> E_1 (E_2)

<u>Ví dụ:</u>

```
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

3.2. Ví dụ về một bộ kiểm tra kiểu đơn giản.

Ví dụ về một ngôn ngữ đơn giản mà kiểu của các biến phải được khai báo trước khi dùng. Bộ kiểm tra kiểu này là một cú pháp dạng lược đồ chuyển đổi nhằm tập hợp kiểu của từng biểu thức từ các kiểu của các biểu thức con. Bộ kiểm tra kiểu có thể làm việc với các mảng, các con trỏ, lệnh, hàm.

* Một văn phạm dưới đây sinh ra các chương trình, biểu diễn bởi biến P, bao gồm một chuỗi các khai báo D theo sau một biểu thức đơn E, các kiểu T.

P $\rightarrow$D;E

D →D;D|**tên**:T

T →**char**| **integer**| **array**| **số**| **of** T| ^T

E →**chữ cái**| **Số | Tên**| E **mod** E | E; E |E^

- Một chương trình có thể sinh ra từ văn phạm trên như sau:

Key: Integer;

Key mod 1999

* Lược đồ chuyển đổi như sau:

P →D; E

D →D;D

D →**Tên**:T *{addtype (tên.entry, T.type)}*

T →**Char** *{T.type:= char}*

T →**integer** *{T.type:= integer}*

T →^T_1 *{T.type:= pointer(T_1.type)}*

T →**array | số | of** T_1 *{T.type:= aray(số.val,T_1.type)}*

*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.*

**** Kiểm tra kiểu của các biểu thức.***

Trong các luật sau:

E →chữ cái {E.type : = char}

E →Số { E.type := integer}

Kiểu của thuộc tính tổng hợp của E cho biểu thưc được gán bằng kiểu hệ thống để sinh biểu thức bởi E. Các luật ngữ nghĩa cho thấy các hằng số biểu diễn bằng từ tố chữ cái và số có kiểu char và integer.

Ta dùng hàm lookup(e) để lấy kiểu caats trong bảng ký hiệu trỏ bởi e. Khi một tên xuất hiện trong biểu thức kiểu khao báo của nó được lấy và gán cho thuộc tính kiểu

E →tên {E.type:= lookup (tên.entry)}

- Một biểu thức được tạo bởi lệnh mod cho 2 biểu thức con có kiểu integer thì nó cũng có kiểu là integer nếu không là kiểu type_error.

E → E_1 mod E_2 {E.type : = if E_1.type = integer and E_2.type = integer then integer else type_error}

- Đối với mảng $E_1[E_2]$bieeur thức chỉ số E_2 phải có kiểu là integer các phần tử của mảng có kiểu t chính là kiểu array(s,t) của E_t

$E \rightarrow E_1[E_2]$ {E.type :=if E_2.type = integer and E_t.type = array(s,t) then t else type_error}

- Đối với thuật toán lấy giá trị con trỏ.

$E \rightarrow E_t$^ {E.type := if E_1.type = pointer (t) then else type_error}

**** Kiểm tra kiểu của câu lệnh:***

Đối với câu lệnh không có giá trị: gán kiểu đặc biệt *void* . nếu có lỗi được phát hiện trong câu lệnh : kiểu câu lệnh là : type_error.

Các câu lệnh gồm: lệnh gán, điều kiện, vòng lặp while. Chuooix các câu lệnh phân cách nhau bằng dấu chấm phẩy. một chương trình hoàn chỉnh có luật dạng P $\rightarrow$ D ; S cho biết một chương trình bao gồm các khai báo và theo sau là các câu lệnh .

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

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

S $\rightarrow$While E do S_1 {S.type:= if E.type = boolean then S_1.type = void then void else type_error }

**** kiểm tra biểu thức của hàm***.

Các hàm với tham số có sản xuất dạng: E $\rightarrow$E (E)

Các luật ứng với kiểu biểu thức của kí hiệu không kết thúc T có thể làm thừa số theo các sản xuất sau:

T $\rightarrow T_1$'$\rightarrow$' T_2 {T.type := T_1.type $\rightarrow T_2$.type}

Luật kiểm tra kiểu của một hàm là: E $\rightarrow E_1(E_2)$ {E.type : =if E_2.type =s $\rightarrow$t then t else type_error}

luật này cho biết trong một biểu thức được tạo bằng cách áp dụng E_1 vào E_2 kiểu của s $\rightarrow$t phải là một hàm từ kiểu của s vào kiểu nào đó t. kiểu $E_1(E_2)$ là t.

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, trớc tiên chương trình dịch đổi cả 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. *(ép kiểu thường gây mất thông tin)*

Một phép đổi kiểu được gọi là rõ nếu người lập trình phải viết số thứ để thực hiện phép đổi này. Ví dụ:

Sản xuất	Luật ngữ nghĩa
E →Số	E.type:= integer
E →Số.số	E.type:= real
E →tên	E.type:= lookup (tên.entry)

$E \rightarrow E_1$ 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 thộc vào ngữ cảnh của nó.

VD: + 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.

CHƯƠNG 7 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 traviệc dùng các tên có khớp với khai báo không) trong giai đoạn sinh mã (kích thước của tên, loại bộ nhớ phải cấp phát cho một tên).

Dùng bảng kí hiệu trong quá trình phát hiện và khôi 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,

Trong bảng ký hiệu cũng có thể có lưu các từ khóa của ngôn ngữ. Nếu vậy thì chúng phải được đưa vào bảng ký hiệu trước khi bộ phân tích từ vựng bắt đầu.

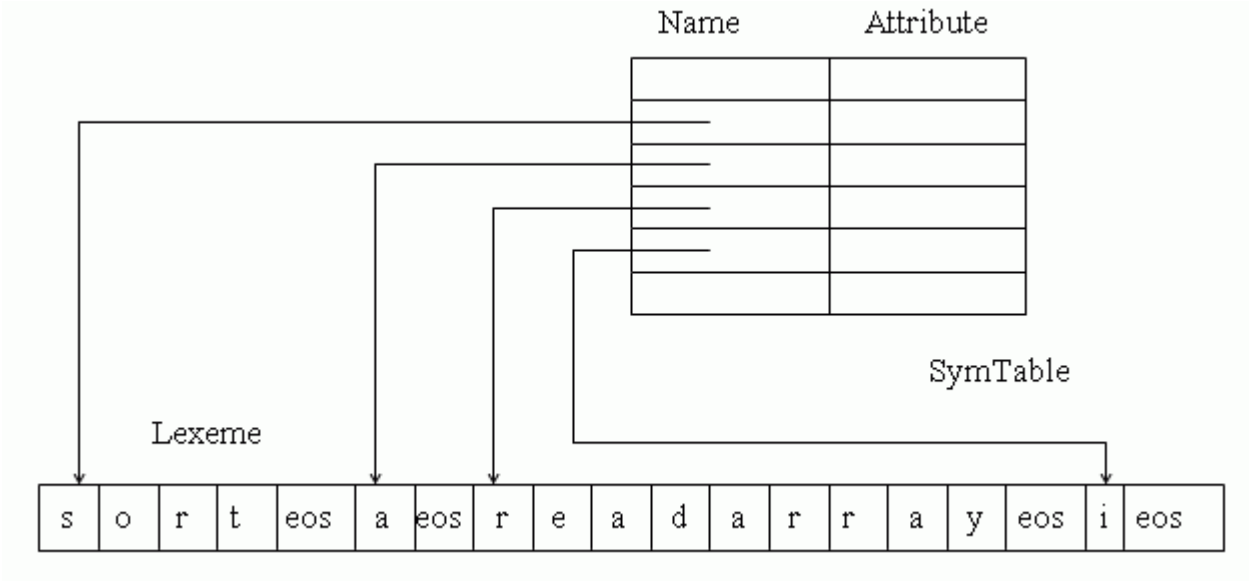
* Nếu ghi trực tiếp tên trong trường tên của bảng thì: ưu điểm: đơn giản, nhanh. Nhược điểm: Độ dài tên bị giới hạn bởi kích thước của trường , hiệu quả sử dụng bộ nhớ không cao.

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 7.19 - *Bảng ký hiệu lưu giữ các tên bị giới hạn độ dài*

Trường hợp độ dài tên không bị giới hạn thì các Lexeme được lưu trong một mảng riêng và bảng ký hiệu chỉ giữ các con trỏ trỏ tới đầu mỗi Lexeme



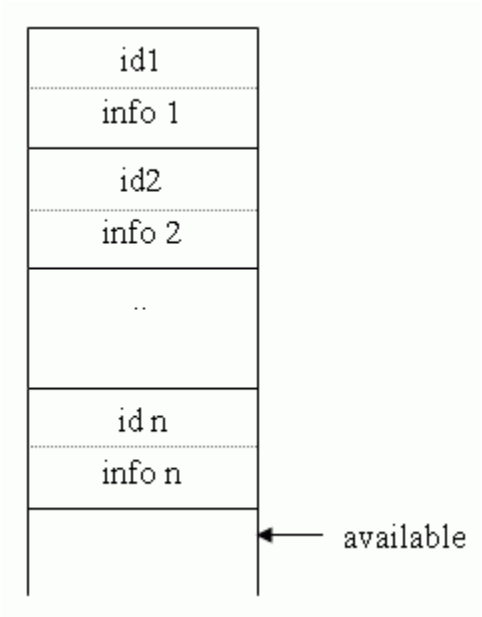
Hình 7.20 - Bảng ký hiệu lưu giữ các tên không bị giới hạn độ dài

3.1 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 7.21 - *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:

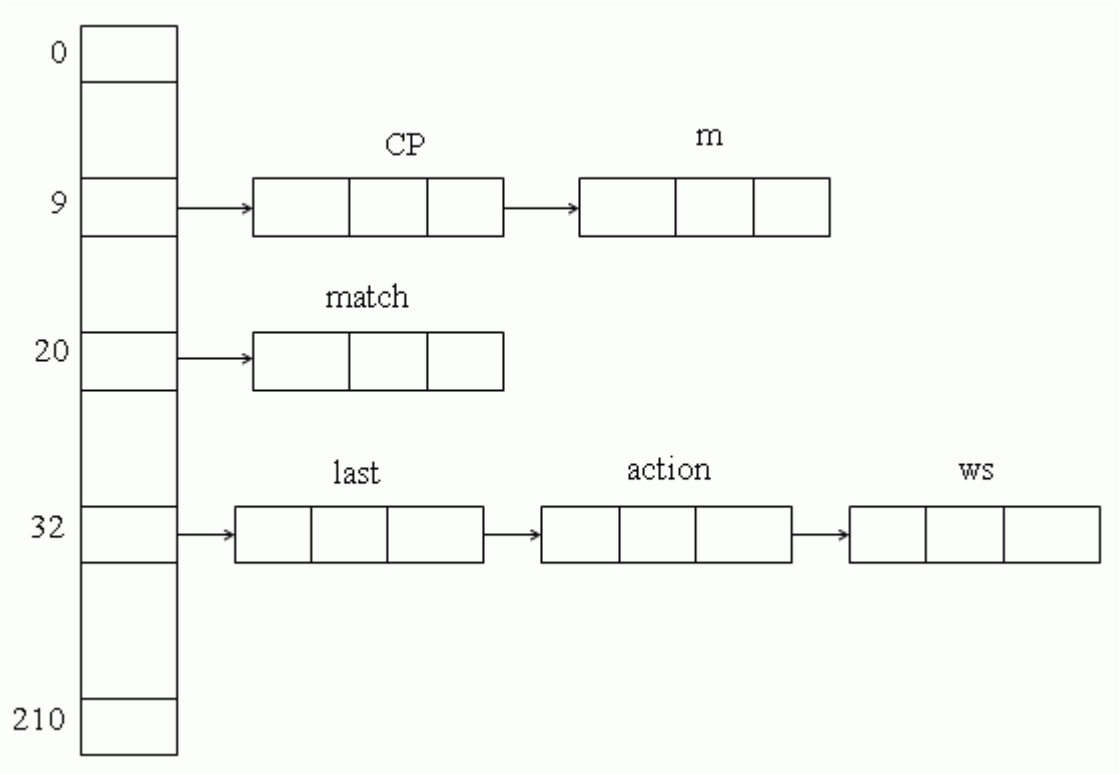
+ Nếu trùng: tìm kiếm thoả mãn.
+ Nếu < hơn: Thực hiện lại cách tìm kiểm với cây con bên trái.
+ Nếu > gốc: thực hiện lại cách tìm kiếm với cây con bên phải.

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 7.22 - *Bảng băm có kích thước 211*

1. Bảng băm là một mảng bao gồm m con trỏ.
2. Bảng danh biểu được chia thành m danh sách liên kết, mỗi danh sách liên kết được trỏ bởi một phần tử trong bảng băm.

Việc phân bổ các danh biểu vào danh sách liên kết nào do hàm băm (hash function) quy định. Giả sử s là chuỗi ký tự xác định danh biểu, hàm băm h tác động lên s trả về một giá trị nằm giữa 0 và m- 1 h(s) = t => Danh biểu s được đưa vào trong danh sách liên kết được trỏ bởi phần tử t của bảng băm.

Có nhiều phương pháp để xác định hàm băm.

Phương pháp đơn giản nhất như sau:

1. Giả sử s bao gồm các ký tự 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 8 SINH MÃ TRUNG GIAN.

1. MỤC ĐÍCH NHIỆM VỤ.

* Sinh mã trung gian có những ưu điểm như sau:

- Dễ thiết kế từng phần

- Sinh được mã độc lập với từng máy tính cụ thể. Từ đó làm giảm độ phức tạp của sinh mã thực sự.

- Dễ tối ưu mã.

* Các vấn đề của bộ sinh mã trung gian là:

  - Dùng mã trung gian nào.

  - Thuật toán sinh mã trung gian.

Hành động sinh mã trung gian thực hiện qua cú pháp điều khiển.

Ngôn ngữ trung gian là ngôn ngữ nằm giữa ngôn ngữ nguồn và ngôn ngữ đích. Chương trình viết bằng ngôn ngữ trung gian vẫn tương đương với chương trình viét bàng ngôn ngữ nguồn về chức năng nhiệm vụ. Sau đây ta xét loại mã trung gian thông dụng nhất.

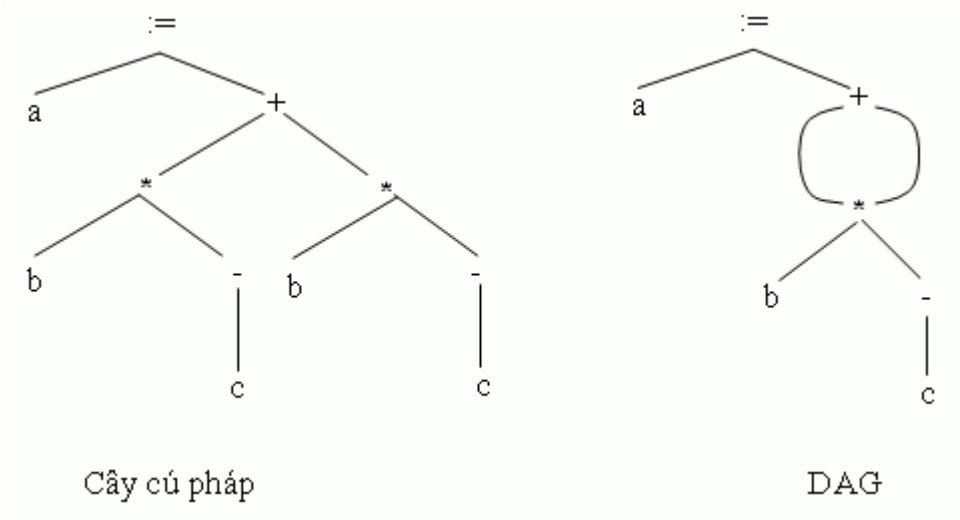
2. CÁC NGÔN NGỮ TRUNG GIAN

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

2.1. 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 trong đó các biểu thức con không được biểu diễn lặp lại.

Ví dụ 8.1: Với lệnh gán a := b * - c + b * - c, ta có cây cú pháp và DAG:



Hình 8.1 - *Biểu diễn đồ thị của a :=b * - c + b * - c*

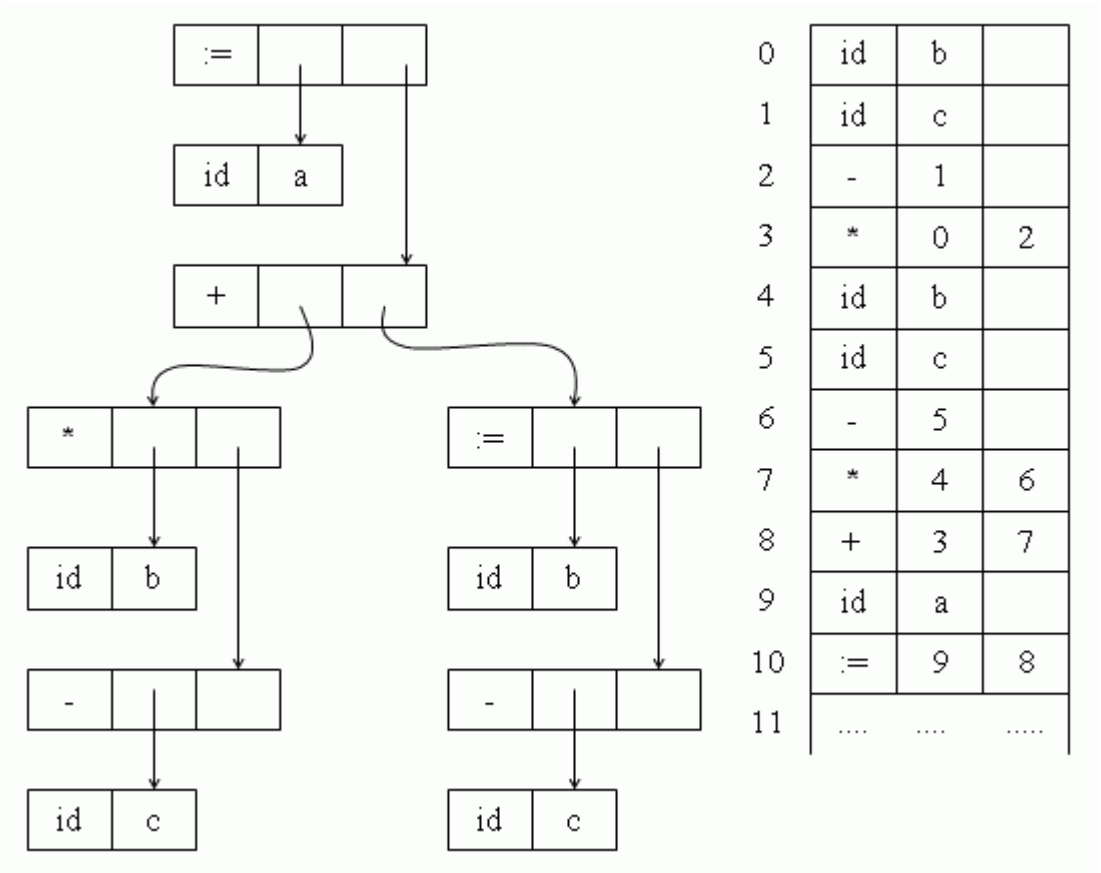
Ký pháp hậu tố là một biểu diễn tuyến tính của cây cú pháp. Nó là một danh sách các nút của cây, trong đó một nút xuất hiện ngay sau con của nó .

a b c - * b c - * + := là biểu diễn hậu tố của cây cú pháp hình trên.

Cây cú pháp có thể được cài đặt bằng một trong 2 phương pháp:

- Mỗi nút được biểu diễn bởi một mẫu tin, với một trường cho toán tử và các trường khác trỏ đến con của nó.
- Một mảng các mẫu tin, trong đó chỉ số của phần tử mảng đóng vai trò như là con trỏ của một nút.

Tất cả các nút trên cây cú pháp có thể tuân theo con trỏ, bắt đầu từ nút gốc tại 10



Hình 8.2 - *Hai biểu diễn của cây cú pháp trong hình 8.1*

2.2. 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:

Ví dụ: Kí pháp hậu tố của (9-5)+2 là 95-2+;

Kí pháp hậu tố của 9-(5+2) là 952+-;

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+?

* Định nghĩa cú pháp điều khiển tạo mã hậu tố.

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…

Dưới đây là 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_2

. . .

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

2.1. Cài đặt các câu lệnh ba địa chỉ

*Trong chương trình dịch, những câu lệnh mã 3 địa chỉ có thể được cài đặt như các cấu trúc với các trường chứa các toán tử và toán hạng. Những biểu diễn đó là bộ tứ (**quadruple**) và bộ ba (**triple**).*

2.1.1. Bộ tứ

Bộ tứ là một cấu trúc bản ghi với bốn trường, được gọi là *op, arg1, arg2* và *result*.

Ví dụ: câu lệnh x := y + z

op là +, arg1 là y, arg2 là z và result chứa x. Đối với toán tử một ngôi thì không dùng arg2.

Ví dụ: Câu lệnh *a := -b * (c+d)*

sẽ được chuyển thành đoạn mã ba địa chỉ như sau:

t1 := - b

t2 := c+d

*t3 := t1 * t2*

a := t3

và được 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

2.1.2. 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)

Chú ý, câu lệnh sao chép đặt kết quả trong arg1 và tham số trong arg2 và 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.

Chú ý, phép toán ba ngôi như x[i] := y cần đến hai mục trong cấu trúc bộ ba như được chỉ ra như sau:

	op	**arg1**	**arg2**
(0)	[]=	x	i
(1)	assign	(0)	y

tương tự đối với phép toán ba ngôi x := y[i]

	op	**arg1**	**arg2**
(0)	[]=	y	i
(1)	assign	x	(0)

2.2. 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.

Sau đây, chúng ta xét ví dụ sinh mã ba địa chỉ cho một số dạng câu lệnh.

2.2.1. 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 := ''

<u>Ví dụ:</u>

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

2.2.2. 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.

<u>Ví dụ:</u> E có dạng *a<b*. Thế thì mã sinh ra sẽ có dạng

if a<b goto E.true

goto E.false

<u>Ví dụ</u> 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.

Để sinh ra các nhãn, chúng ta sử dụng thủ tục newlable để sinh ra một nhãn mới.

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)

<u>Ví dụ:</u> Sinh mã ba địa chỉ cho đoạn chương trình sau:

if a>b and c>d then

x:=y+z

else

x:=y-z

<u>Lời giải:</u>

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
```

Như vậy chúng ta phải phân tích bên trong của biểu thức E, và dễ thấy các lệnh nhảy bên trong E chính là E.true và E.false, điều đó giải thích tại sao chúng ta lại có các luật ngữ nghĩa như bảng trên.

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:
```

2.2.3. Sinh mã ba địa chỉ cho một số lệnh điều khiển:

Trong các câu lệnh điều khiển có điều kiện, ta dựa vào biểu thức logic E để chuyển việc thực hiện các câu lệnh tới vị trí thích hợp. Do đó ta cần hai nhãn: *E.true* (*để xác định vị trí câu lệnh chuyển tới khi biểu thức logic E là đúng*), nhãn *E.false* (*để xác định vị trí câu lệnh chuyển tới khi biểu thức logic E là sai*).

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

Nhãn *S.next* đối với khối lệnh sinh ra bởi ký hiệu S là nhãn xác định vị trí tiếp theo của các lệnh sau S.

Đối với câu lệnh *S -> while E do S_1*

ta cần có một nhãn bắt đầu của khối lệnh này để nhảy đến mỗi khi E đúng, vì vậy cần nhãn *S.begin* để xác định vị trí bắt đầu khối lệnh này.

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)

<u>Ví dụ 1:</u> sinh đoạn mã ba địa chỉ cho đoạn mã nguồn sau:

```
while a<>b do
    if a>b then
    a:=a-b
    else
      b:=b-a
```

<u>Lời giải:</u>

```
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:
```

2.2.3.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.

<u>Ví dụ:</u>

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. *Hay nói cách khác chúng ta có thể thay một định danh bởi chỉ mục của định danh đó trong bảng ký hiệu.*

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

Bài tập 1: Hãy chuyển các câu lệnh hoặc đoạn chương trình sau thành đoạn 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;
  }
}
```

.1. Dịch biểu thức : **a * - (b + c)** thành các dạng :

a) Cây cú pháp.
b) Ký pháp hậu tố.
c) Mã lệnh máy 3 - địa chỉ.

8.2. Trình bày cấu trúc lưu trữ biểu thức **- (a + b) * (c + d) + (a + b + c)** ở các dạng :

a) Bộ tứ .
b) Bộ tam.
c) Bộ tam gián tiếp.

8.3. Sinh mã trung gian (dạng mã máy 3 - địa chỉ) cho các biểu thức C đơn giản sau :

a) x = 1
b) x = y
c) x = x + 1
d) x = a + b * c
e) x = a / (b + c) - d * (e + f)

8.4. Sinh mã trung gian (dạng mã máy 3 - địa chỉ) cho các biểu thức C sau :

a) x = a [i] + 11
b) a [i] = b [c[j]]
c) a [i][j] = b [i][k] * c [k][j]
d) a[i] = a[i] + b[j]
e) a[i] + = b[j]

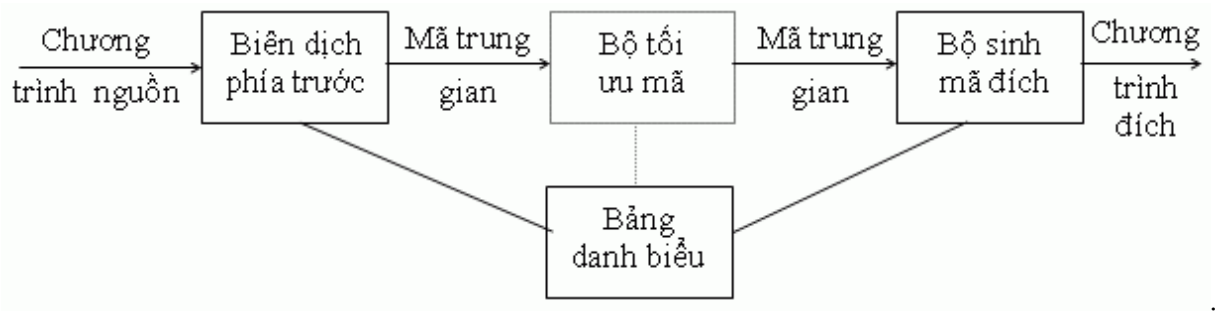
8.5. Dịch lệnh gán sau thành mã máy 3 - địa chỉ :

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

CHƯƠNG 9 SINH MÃ

1. MỤC ĐÍCH NHIỆM VỤ

Giai đoạn cuối của quá trình biên dịch là sinh mã đích. Kỹ thuật sinh mã đích được trình bày trong chương này không phụ thuộc vào việc dùng hay không dùng giai đoạn tối ưu mã trung gian



Sinh mã tốt rất khó, mã sinh ra thường gắn với một loại máy tính cụ thể nào đó.

Đầu vào của bộ sinh mã là mã trung gian, đầu ra là một chương trình viết dạng mã đối tượng nào đó và gọi là chương trình đích.

Đầu vào của bộ sinh mã gồm biểu diễn trung gian của chương trình nguồn, cùng thông tin trong bảng danh biểu được dùng để xác định địa chỉ của các đối tượng dữ liệu trong thời gian thực thi. Các đối tượng dữ liệu này được tượng trưng bằng tên trong biểu diễn trung gian. Biểu diễn trung gian của chương trình nguồn có thể ở một trong các dạng: ký pháp hậu tố, mã ba địa chỉ, cây cú pháp, DAG

Tiêu chuẩn quan trọng nhất đối với bộ sinh mã là sinh mã đúng.Tính đúng của mã có một ý nghĩa rất quan trọng. Với những quy định về tính đúng của mã, việc thiết kế bộ sinh mã sao cho nó được thực hiện, kiểm tra, bảo trì đơn giản là mục tiêu thiết kế quan trọng .

2. CÁC DẠNG MÃ ĐỐI TƯỢNG.

2.1. Mã máy định vị tuyệt đối.

Một chương trình mã máy tuyệt đối có các lệnh mã máy được định vị tuyệt đối. Chương trình dịch xác định hoàn toàn chương trình đối tượng này.

Mã được một chương trình dịch thực sự tạo ra và đặt vào các vị trí này nên chương trình có thể hoạt động ngay.

Ưu điểm: giảm số

2.2. Mã đối tượng có thể định vị lại được.

2.3. Mã đối tượng thông dịch.

Việc tạo ra chương đích ở dạng hợp ngữ cho phép ta dùng bộ biên dịch hợp ngữ để tạo ra mã máy.

3. CÁC VẤN ĐỀ THIẾT KẾ CỦA BỘ SINH MÃ.

Sự lựa chọn chỉ thị

Tập các chỉ thị của máy đích sẽ xác định tính phức tạp của việc lựa chọn chỉ thị. Tính chuẩn và hoàn chỉnh của tập chỉ thị là những yếu tố quan trọng. Nếu máy đích không cung cấp một mẫu chung cho mỗi kiểu dữ liệu thì mỗi trường hợp ngoại lệ phải xử lý riêng. Tốc độ chỉ thị và sự biểu diễn của máy cũng là những yếu tố quan trọng. Nếu ta không quan tâm đến tính hiệu quả của chương trình đích thì việc lựa chọn chỉ thị sẽ đơn giản hơn. Với mỗi lệnh ba địa chỉ ta có thể phác họa một bộ khung cho mã đích. Giả sử lệnh ba địa chỉ dạng x := y + z, với x, y, z được cấp phát tĩnh, có thể được dịch sang chuỗi mã đích:

```
MOV  y, R0      /* Lưu y vào thanh ghi Ro */
ADD  z, R0      /* cộng z vào nội dung Ro, kết quả chứa trong Ro */
MOV  R0, x      /* lưu nội dung Ro vào x */
```

Tuy nhiên việc sinh mã cho chuỗi các lệnh ba địa chỉ sẽ dẫn đến sự dư thừa mã. Chẳng hạn với:

a:= b + c
d:= a + e

ta chuyển sang mã đích:

MOV b, R_o
ADD c, R_o
MOV R_o, a
MOV a, R_0
ADD e,R_o
MOV R_o, d

và ta nhận thấy rằng chỉ thị thứ tư là thừa.

Chất lượng mã được tạo ra, được xác định bằng tốc độ và kích thước của mã. Một máy đích có tập chỉ thị phong phú có thể sẽ cung cấp nhiều cách để hiện thực một tác vụ cho trước. Điều này có thể dẫn đến tốc độ thực hiện chỉ thị rất khác nhau. Chẳng hạn, nếu máy

đích có chỉ thị INC thì câu lệnh ba địa chỉ a *:= a + 1 có thể được cài đặt chỉ bằng câu lệnh INC a. Cách nầy hiệu quả* hơn là dùng chuỗi các chỉ thị sau:

MOV a, R_o
ADD # 1, R_o
MOV R_o,a

Như ta đã nói, tốc độ của chỉ thị là một trong những yếu tố quan trọng để thiết kế chuỗi mã tốt. Nhưng, thông tin thời gian thường khó xác định.

Việc quyết định chuỗi mã máy nào là tốt nhất cho câu lệnh ba địa chỉ còn phụ thuộc vào ngữ cảnh của nơi chứa câu lệnh đó.

Cấp phát thanh ghi

Các chỉ thị dùng toán hạng thanh ghi thường ngắn hơn và nhanh hơn các chỉ thị dùng toán hạng trong bộ nhớ. Vì thế, hiệu quả của thanh ghi đặc biệt quan trọng trong việc sinh mã tốt. Ta thường dùng thanh ghi trong hai trường hợp:

1. Trong khi cấp phát thanh ghi, ta lựa chọn tập các biến lưu trú trong các thanh ghi tại một thời điểm trong chương trình.
2. Trong khi gán thanh ghi, ta lấy ra thanh ghi đặc biệt mà biến sẽ thường trú trong đó.

Việc tìm kiếm một lệnh gán tối ưu của thanh ghi, ngay với cả các giá trị thanh ghi đơn, cho các biến là một công việc khó khăn. Vấn đề càng trở nên phức tạp hơn vì phần cứng và / hoặc hệ điều hành của máy đích yêu cầu qui ước sử dụng thanh ghi.

3.3. Quản lý bộ nhớ.

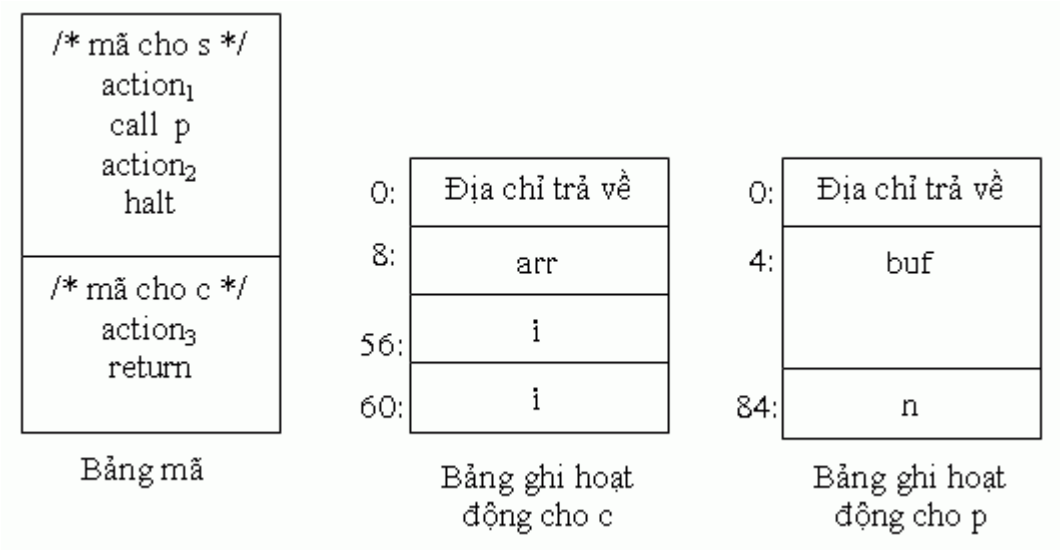
Trong phần này ta sẽ nói về việc sinh mã để quản lý các mẩu tin hoạt động trong thời gian thực hiện. Hai chiến lược cấp phát bộ nhớ chuẩn được trình bầy trong chương VII là cấp phát tĩnh và cấp phát Stack. Với cấp phát tĩnh, vị trí của mẩu tin hoạt động trong bộ nhớ được xác định trong thời gian biên dịch. Với cấp phát Stack, một mẩu tin hoạt động được đưa vào Stack khi có sự thực hiện một thủ tục và được lấy ra khỏi Stack khi hoạt động kết thúc. Ở đây, ta sẽ xem xét cách thức mã đích của một thủ tục tham chiếu tới các đối tượng dữ liệu trong các mẩu tin hoạt động. Như ta đã nói ở chương VII, một mẩu tin hoạt động cho một thủ tục có các trường: tham số, kết quả, thông tin về trạng thái máy, dữ liệu cục bộ, lưu trữ tạm thời và cục bộ, và các liên kết. Trong phần nầy, ta minh họa các chiến lược cấp phát sử dụng trường trạng thái để giữ giá trị trả về và dữ liệu cục bộ, các trường còn lại được dùng như đã đề cập ở chương VII.

Việc cấp phát và giải phóng các mẩu tin hoạt động là một phần trong chuỗi hành vi gọi và trả về của chương trình con. Ta quan tâm đến việc sinh mã cho các lệnh sau:

1. call
2. return
3. halt

4. action / tượng trưng cho các lệnh khác */*

Chẳng hạn, mã ba địa chỉ, chỉ chứa các loại câu lệnh trên, cho các chương trình c và p cũng như các mẩu tin hoạt động của chúng:



Hình 9.2 - *Đầu vào của bộ sinh mã*

Kích thước và việc xếp đặt các mẩu tin được kết hợp với bộ sinh mã nhờ thông tin về tên trong bảng danh biểu.

Ta giả sử bộ nhớ thời gian thực hiện được phân chia thành các vùng cho mã, dữ liệu tĩnh và Stack.

1. Cấp phát tĩnh

Chúng ta sẽ xét các chỉ thị cần thiết để thực hiện việc cấp phát tĩnh. Lệnh call trong mã trung gian được thực hiện bằng dãy hai chỉ thị đích. Chỉ thị MOV lưu địa chỉ trả về. Chỉ thị GOTO chuyển quyền điều khiển cho chương trình được gọi.

MOV # here + 20, callee.static_area
GOTO callee.code_area

Các thuộc tính callee.static_area và callee.code_area là các hằng tham chiếu tới các địa chỉ của mẩu tin hoạt động và chỉ thị đầu tiên trong đoạn mã của chương trình con được gọi. # here + 20 trong chỉ thị MOV là địa chỉ trả về. Nó cũng chính là địa chỉ của chỉ thị đứng sau lệnh GOTO. Mã của chương trình con kết thúc bằng lệnh trả về chương trình gọi, trừ chương trình chính, đó là lệnh halt. Lệnh này trả quyền điều khiển cho hệ điều hành. Lệnh trả về được dịch sang mã máy là *GOTO *callee_static_area* thực hiện việc chuyển quyền điều khiển về địa chỉ được lưu giữ ở ô nhớ đầu tiên của mẩu tin hoạt động .

Ví dụ 9.1: Mã đích trong chương trình sau được tạo ra từ các chương trình con c và p ở hình 9.2. Giả sử rằng: các mã đó được lưu tại địa chỉ bắt đầu là 100 và 200, mỗi chỉ thị action chiếm 20 byte, và các mẩu tin hoạt động cho c và p được cấp phát tĩnh bắt đầu tại các địa chỉ 300 và 364 . Ta dùng chỉ thị action để thực hiện câu lệnh action. Như vậy, mã đích cho các chương trình con:

```
                                    /* mã cho c*/
100: ACTION1
120: MOV   #140, 364        /* lưu địa chỉ trả về 140 */
132: GOTO   200             /* gọi p */
140: ACTION2
160: HALT

                                    /* mã cho p */
200: ACTION3
220: GOTO   *364            /* trả về địa chỉ được lưu tại vị trí 364 */
                                    /* 300-364 lưu mẩu tin hoạt động của c */
300:                        /* chứa địa chỉ trả về */
304:                        /* dữ liệu cục bộ của c */
                                    /* 364 - 451 chứa mẩu tin hoạt động của p */
364:                        /* chứa địa chỉ trả về */
368:                        /* dữ liệu cục bộ của p */
```

Hình 9.3 - *Mã đích cho đầu vào của hình 9.2*

Sự thực hiện bắt đầu bằng chỉ thị action tại địa chỉ 100. Chỉ thị MOV ở địa chỉ 120 sẽ lưu địa chỉ trả về 140 vào trường trạng thái máy, là từ đầu tiên trong mẩu tin hoạt động của p. Chỉ thị GOTO 200 sẽ chuyển quyền điều khiển về chỉ thị đầu tiên trong đoạn mã của chương trình con p. Chỉ thị GOTO *364 tại địa chỉ 132 chuyển quyền điều khiển sang chỉ thị đầu tiên trong mã đích của chương trình con được gọi.

Giá trị 140 được lưu vào địa chỉ 364, *364 biểu diễn giá trị 140 khi lệnh GOTO tại địa chỉ 220 được thực hiện. Vì thế quyền điều khiển trả về địa chỉ 140 và tiếp tục thực hiện chương trình con c.

2. Cấp phát theo cơ chế Stack

Cấp phát tĩnh sẽ trở thành cấp phát Stack nếu ta sử dụng địa chỉ tương đối để lưu giữ các mẩu tin hoạt động. Vị trí mẩu tin hoạt động chỉ được xác định trong thời gian thực thi. Trong cấp phát Stack, vị trí nầy thường được lưu vào thanh ghi. Vì thế các ô nhớ của mẩu tin hoạt động được truy xuất như là độ dời (offset) so với giá trị trong thanh ghi đó.

Thanh ghi SP chứa địa chỉ bắt đầu của mẩu tin hoạt động của chương trình con nằm trên đỉnh Stack. Khi lời gọi của chương trình con xuất hiện, chương trình bị gọi được cấp

phát, SP được tăng lên một giá trị bằng kích thước mẩu tin hoạt động của chương trình gọi và chuyển quyền điều khiển cho chương trình con được gọi. Khi quyền điều khiển trả về cho chương trình gọi, SP giảm đi một khoảng bằng kích thước mẩu tin hoạt động của chương trình gọi. Vì thế, mẩu tin của chương trình con được gọi đã được giải phóng.

Mã cho chương trình con đầu tiên có dạng:

```
MOV   # Stackstart, SP          /* khởi động Stack */
Đoạn mã cho chương trình con
HALT                             /* kết thúc sự thực thi */
```

Trong đó chỉ thị đầu tiên MOV #Stackstart, SP khởi động Stack theo cách đặt SP bằng với địa chỉ bắt đầu của Stack trong vùng nhớ.

Chuỗi gọi sẽ tăng giá trị của SP, lưu giữ địa chỉ trả về và chuyển quyền điều khiển về chương trình được gọi.

```
ADD   # caller.recordsize, SP
MOV   # here + 16, *SP          /* lưu địa chỉ trả về */
GOTO   callee.code_area
```

Thuộc tính caller.recordsize biểu diễn kích thước của mẩu tin hoạt động. Vì thế, chỉ thị ADD đưa SP trỏ tới phần bắt đầu của mẩu tin hoạt động kế tiếp. #here +16 trong chỉ thị MOV là địa chỉ của chỉ thị theo sau GOTO, nó được lưu tại địa chỉ được trỏ bởi SP.

Chuỗi trả về gồm hai chỉ thị:

1. Chương trình con chuyển quyền điều khiển tới địa chỉ trả về

```
GOTO   *0(SP)                    /* trả về chương trình gọi */
SUB   #caller.recordsize,  SP
```

Trong đó O(SP) là địa chỉ của ô nhớ đầu tiên trong mẩu tin hoạt động. *O(SP) trả về địa chỉ được lưu tại đây.

2. Chỉ thị SUB #caller.recordsize, SP: Giảm giá trị của SP xuống một khoảng bằng kích thước mẩu tin hoạt động của chương trình gọi. Như vậy mẩu tin hoạt động chương trình bị gọi đã xóa khỏi Stack .

Ví dụ 9.2: Giả sử rằng kích thước của các mẩu tin hoạt động của các chương trình con s, p, và q được xác định tại thời gian biên dịch là ssize, psize, và qsize tương ứng. Ô nhớ đầu tiên trong mỗi mẩu tin hoạt động lưu địa chỉ trả về. Ta cũng giả sử rằng, đoạn mã cho các chương trình con nầy bắt đầu tại các địa chỉ 100, 200, 300 tương ứng, và địa chỉ bắt đầu của Stack là 600. Mã đích cho chương trình trong hình 9.4 được mô tả trong hình 9.5:

/* mã cho s */ action1 call q action2 halt
/* mã cho p */ action3 return
/* mã cho q */ $action_4$ call p $action_5$ call q $action_6$ call q return

Hình 9.4 - *Mã ba địa chỉ minh hoạ cấp phát sử dụng Stack*

/* mã cho s*/

100: MOV # 600, SP /* khởi động Stack */
108: ACTION$_1$
128: ADD #ssize, SP /* chuỗi gọi bắt đầu */
*136: MOV #152, *SP* /* lưu địa chỉ trả về */
144: GOTO 300 /* gọi q */
152: SUB #ssize, SP /* Lưu giữ SP */
160: ACTION$_2$
180: HALT

/* mã cho p */

200: ACTION$_3$
*220: GOTO *0(SP)* /* trả về chương trình gọi */

/* mã cho q */

300: ACTION4 /* nhảy có điều kiện về 456 */
320: ADD #qsize, SP
*328: MOV #344, *SP* /* lưu địa chỉ trả về */
336: GOTO 200 /* gọi p */
344: SUB #qsize, SP
352: ACTION$_5$
372: ADD #qsize, SP
*380: MOV #396, *SP* /* lưu địa chỉ trả về */
388: GOTO 300 /* gọi q */

```
396: SUB   #qsize, SP
404: ACTION6
424: ADD   #qsize, SP
432: MOV   #448, *SP          /* lưu địa chỉ trả về */
440: GOTO   300               /* gọi q */
448: SUB   #qsize, SP
456: GOTO   *0(SP)            /* trả về chương trình gọi */

600:                          /* địa chỉ bắt đầu của Stack trung tâm */
```

Hình 9.5 - *Mã đích cho chuỗi ba địa chỉ trong hình 9.4*

Ta giả sử rằng action4 gồm lệnh nhảy có điều kiện tới địa chỉ 456 có lệnh trả về từ q. Ngược lại chương trình đệ quy q có thể gọi chính nó mãi. Trong ví dụ này chúng ta giả sử lần gọi đầu tiên trên q sẽ không trả về chương tình gọi ngay, nhưng những lần sau thì có thể. SP có giá trị lúc đầu là 600, địa chỉ bắt đầu của Stack. SP lưu giữ giá trị 620 chỉ trước khi chuyển quyền điều khiển từ s sang q vì kích thước của mẩu tin hoạt động s là 20. Khi q gọi p, SP sẽ tăng lên 680 khi chỉ thị tại địa chỉ 320 được thực hiện, Sp chuyển sang 620 sau khi chuyển quyền điều khiển cho chương trình con p. Nếu lời gọi đệ quy của q trả về ngay thì giá trị lain nhất của SP trong suốt quá trình thực hiện là 680. Vị trí được cấp phát theo cơ chế Stack có thể lên đến địa chỉ 739 vì mẩu tin hoạt động của q bắt đầu tại 680 và chiếm 60 byte.

3. Địa chỉ của các tên trong thời gian thực hiện

Chiến lược cấp phát lưu trữ và xếp đặt dữ liệu cục bộ trong mẩu tin hoạt động của chương trình con xác định cách thức truy xuất vùng nhớ của tên.

Nếu chúng ta dùng cơ chế cấp phát tĩnh với vùng dữ liệu được cấp phát tại địa chỉ static. Với lệnh gán x := 0, địa chỉ tương đối của x trong bảng danh biểu là 12. Vậy địa chỉ của x trong bộ nhớ là static + 12. Lệnh gán x:=0 được chuyển sang mã ba địa chỉ static[12] := 0. Nếu vùng dữ liệu bắt đầu tại địa chỉ 100, mã đích cho chỉ thị là:

```
MOV  #0,112
```

Nếu ngôn ngữ dùng cơ chế display để truy xuất tên không cục bộ, giả sử x là tên cục bộ của chương trình con hiện hành và thanh ghi R3 lưu giữ địa chỉ bắt đầu của mẩu tin hoạt động đó thì chúng ta sẽ dịch lệnh x := 0 sang chuỗi mã ba địa chỉ:

$t_1 := 12 + R_3$
$* t_1 := 0$

Từ đó ta chuyển sang mã đích:

$MOV \quad \#0, 12(R_3)$

Chú ý rằng, giá trị thanh ghi R3 không được xác định trong thời gian biên dịch

3.4. Chọn chỉ thị lệnh.

Tập các chỉ thị của máy đích sẽ xác định tính phức tạp của việc lựa chọn chỉ thị. Tính chuẩn và hoàn chỉnh của tập chỉ thị là những yếu tố quan trọng. Nếu máy đích không cung cấp một mẫu chung cho mỗi kiểu dữ liệu thì mỗi trường hợp ngoại lệ phải xử lý riêng. Tốc độ chỉ thị và sự biểu diễn của máy cũng là những yếu tố quan trọng. Nếu ta không quan tâm đến tính hiệu quả của chương trình đích thì việc lựa chọn chỉ thị sẽ đơn giản hơn. Với mỗi lệnh ba địa chỉ ta có thể phác họa một bộ khung cho mã đích. Giả sử lệnh ba địa chỉ dạng x := y + z, với x, y, z được cấp phát tĩnh, có thể được dịch sang chuỗi mã đích:

```
MOV  y, R0      /* Lưu y vào thanh ghi Ro */
ADD  z, R0      /* cộng z vào nội dung Ro, kết quả chứa trong Ro */
MOV  R0, x      /* lưu nội dung Ro vào x */
```

Tuy nhiên việc sinh mã cho chuỗi các lệnh ba địa chỉ sẽ dẫn đến sự dư thừa mã. Chẳng hạn với:

a:= b + c
d:= a + e

ta chuyển sang mã đích:

```
MOV  b, Ro
ADD  c, Ro
MOV  Ro, a
MOV  a, R0
ADD  e,Ro
MOV  Ro, d
```

và ta nhận thấy rằng chỉ thị thứ tư là thừa.

Chất lượng mã được tạo ra, được xác định bằng tốc độ và kích thước của mã. Một máy đích có tập chỉ thị phong phú có thể sẽ cung cấp nhiều cách để hiện thực một tác vụ cho trước. Điều này có thể dẫn đến tốc độ thực hiện chỉ thị rất khác nhau. Chẳng hạn, nếu máy đích có chỉ thị INC thì câu lệnh ba địa chỉ a *:= a + 1 có thể được cài đặt chỉ bằng câu lệnh INC a. Cách nầy hiệu quả* hơn là dùng chuỗi các chỉ thị sau:

```
MOV  a, Ro
ADD  # 1, Ro
MOV  Ro ,a
```

Như ta đã nói, tốc độ của chỉ thị là một trong những yếu tố quan trọng để thiết kế chuỗi mã tốt. Nhưng, thông tin thời gian thường khó xác định.

Việc quyết định chuỗi mã máy nào là tốt nhất cho câu lệnh ba điạ chỉ còn phụ thuộc vào ngữ cảnh của nơi chưá câu lệnh đó.

3.5. Sử dụng thanh ghi.

Các chỉ thị dùng toán hạng thanh ghi thường ngắn hơn và nhanh hơn các chỉ thị dùng toán hạng trong bộ nhớ. Vì thế, hiệu quả của thanh ghi đặc biệt quan trọng trong việc sinh mã tốt. Ta thường dùng thanh ghi trong hai trường hợp:

1. Trong khi cấp phát thanh ghi, ta lựa chọn tập các biến lưu trú trong các thanh ghi tại một thời điểm trong chương trình.
2. Trong khi gán thanh ghi, ta lấy ra thanh ghi đặc biệt mà biến sẽ thường trú trong đó.

Việc tìm kiếm một lệnh gán tối ưu của thanh ghi, ngay với cả các giá trị thanh ghi đơn, cho các biến là một công việc khó khăn. Vấn đề càng trở nên phức tạp hơn vì phần cứng và / hoặc hệ điều hành của máy đích yêu cầu qui ước sử dụng thanh ghi.

3.6. Thứ tự làm việc.

Thứ tự thực hiện tính toán có thể ảnh hưởng đến tính hiệu quả của mã đích . Một số thứ tự tính toán có thể cần ít thanh ghi để lưu giữ các kết quả trung gian hơn các thứ tự tính toán khác. Việc lựa chọn được thứ tự tốt nhất là một vấn đề khó. Ta nên tránh vấn đề này bằng cách sinh mã cho các lệnh ba địa chỉ theo thứ tự mà chúng đã được sinh ra bởi bộ mã trung gian.

Sinh mã

Tiêu chuẩn quan trọng nhất của bộ sinh mã là phải tạo ra mã đúng. Tính đúng của mã có một ý nghĩa rất quan trọng. Với những quy định về tính đúng của mã, việc thiết kế bộ sinh mã sao cho nó được thực hiện, kiểm tra, bảo trì đơn giản là mục tiêu thiết kế quan trọng .

4. MÁY ĐÍCH.

Trong chương trình này, chúng ta sẽ dùng máy đích như là máy thanh ghi (register machine). Máy này tượng trưng cho máy tính loại trung bình. Tuy nhiên, các kỹ thuật sinh mã được trình bầy trong chương này có thể dùng cho nhiều loại máy tính khác nhau.

Máy đích của chúng ta là máy tính địa chỉ byte với mỗi từ gồm bốn byte và có n thanh ghi : R_0, R_1 ... R_{n-1} . Máy đích gồm các chỉ thị hai địa chỉ có dạng chung:

op source, destination

Trong đó op là mã tác vụ. Source (nguồn) và destination (đích) là các trường dữ liệu. Ví dụ một số mã tác vụ:

MOV *chuyển source đến destination*
ADD *cộng source và destination*
SUB *trừ source cho destination*

Source và destination của một chỉ thị được xác định bằng cách kết hợp các thanh ghi và các vị trí nhớ với các mode địa chỉ. Mô tả content (a) biểu diễn cho nội dung của thanh ghi hoặc địa chỉ của bộ nhớ được biểu diễn bởi a.

mode địa chỉ cùng với dạng hợp ngữ và giá kết hợp:

Mode	Dạng	Địa chỉ	Giá
Absolute	*M*	*M*	*1*
Register	*R*	*R*	*0*
Indexed	*c(R)*	*c + contents (R)*	*1*
Indirect register	**R*	*contents (R)*	*0*
Indirect indexed	**c(R)*	*contents (c+ contents (R))*	*1*

Vị trí nhớ M hoặc thanh ghi R biểu diễn chính nó khi được sử dụng như một nguồn hay đích. Độ dời địa chỉ c từ giá trị trong thanh ghi R được viết là c(R).

Chẳng hạn:

1. MOV R_0, M : Lưu nội dung của thanh ghi R_0 vào vị trí nhớ M .
2. MOV 4(R_0), M : Xác định một địa chỉ mới bằng cách lấy độ dời tương đối (offset) 4 cộng với nội dung của R_0, sau đó lấy nội dung tại địa chỉ này, contains(4 + contains(R_0)), lưu vào vị trí nhớ M.
3. MOV * 4(R_0) , M : Lưu giá trị contents (contents (4 + contents (R_0))) vào vị trí nhớ M.
4. MOV #1, R_0 : Lấy hằng 1 lưu vào thanh ghi R_0.

Giá của chỉ thị

Giá của chỉ thị (instrustion cost) được tính bằng một cộng với giá kết hợp mode địa chỉ nguồn và đích trong bảng trên. Giá này tượng trưng cho chiều dài của chỉ thị. Mode địa chỉ dùng thanh ghi sẽ có giá bằng không và có giá bằng một khi nó dùng vị trí nhớ hoặc hằng. Nếu vấn đề vị trí nhớ là quan trọng thì chúng ta nên tối thiểu hóa chiều dài chỉ thị. Đối với phần lớn các máy và phần lớn các chỉ thị, thời gian cần để lấy một chỉ thị từ bộ nhớ bao giờ cũng xảy ra trước thời gian thực hiện chỉ thị. Vì vậy, bằng việc tối thiểu hóa độ dài chỉ thị, ta còn tối thiểu hoá được thời gian cần để thực hiện chỉ thị.

Một số minh họa việc tính giá của chỉ thị:

1. Chỉ thị MOV R_0, R_1 : Sao chép nội dung thanh ghi R_0 vào thanh ghi R_1. Chỉ thị này có giá là một vì nó chỉ chiếm một từ trong bộ nhớ .

2. MOV R_5, M: Sao chép nội dung thanh ghi R_5 vào vị trí nhớ M. Chỉ thị này có giá trị là hai vì địa chỉ của vị trí nhớ M là một từ sau chỉ thị.

3. Chỉ thị ADD #1, R_3: cộng hằng 1 vào nội dung thanh ghi R_3. Chỉ thị có giá là hai vì hằng 1 phải xuất hiện trong từ kế tiếp sau chỉ thị.

4. Chỉ thị SUB 4(R_0), *12 (R_1) : Lưu giá trị của contents (contents (12 + contents (R1))) - contents (4 + contents (R_0)) vào đích *12(R_1). Giá của chỉ thị nầy là ba vì hằng 4 và 12 được lưu trữ trong hai từ kế tiếp theo sau chỉ thị.

Với mỗi câu lệnh ba địa chỉ, ta có thể có nhiều cách cài đặt khác nhau. Ví dụ câu lệnh a := b + c - trong đó b và c là biến đơn, được lưu chứa trong các vị trí nhớ phân biệt có tên b, c - có những cách cài đặt sau:

1. *MOV b, R_o*
ADD c, R0 *giá = 6*
MOV R_o, a

2. *MOV b, a* *giá = 6*
ADD c, a

3. Giả sử thanh ghi R0, R1, R2 giữ địa chỉ của a, b, c. Chúng ta có thể dùng hai địa chỉ sau cho việc sinh mã lệnh:

a := b + c =>
*MOV *R1, *Ro* *giá = 2*
*ADD * R_2, *R_o*

4. Giả sử thanh ghi R1 và R2 chứa giá trị của b và c và trị của b không cần lưu lại sau lệnh gán. Chúng ta có thể dùng hai chỉ thị sau:

ADD R2, R1 *giá = 3*
MOV R_1, a

Như vậy, với mỗi cách cài đặt khác nhau ta có những giá khác nhau. Ta cũng thấy rằng muốn sinh mã tốt thì phải hạ giá của các chỉ thị . Tuy nhiên việc làm khó mà thực hiện được. Nếu có những quy ước trước cho thanh ghi, lưu giữ địa chỉ của vị trí nhớ chứa giá trị tính toán hay địa chỉ để đưa trị vào, thì việc lựa chọn chỉ thị sẽ dễ dàng hơn.

5. MỘT BỘ SINH MÃ ĐƠN GIẢN.

Ta giả sử rằng, bộ sinh mã này sinh mã đích từ chuỗi các lệnh ba địa chỉ. Mỗi toán tử trong lệnh ba địa chỉ tương ứng với một toán tử của máy đích. Các kết quả tính toán có thể nằm lại trong thanh ghi cho tới bao lâu có thể được và chỉ được lưu trữ khi:

(a) Thanh ghi đó được sử dụng cho sự tính toán khác
(b) Trước khi có lệnh gọi chương trình con, lệnh nhảy hoặc lệnh có nhãn.

Điều kiện (b) chỉ ra rằng bất cứ giá trị nào cũng phải được lưu vào bộ nhớ trước khi kết thúc một khối cơ bản. Vì sau khi ra khỏi khối cơ bản, ta có thể đi tới các khối khác hoặc ta có thể đi tới một khối xác định từ một khối khác. Trong trường hợp (a), ta không

thể làm được điều nầy mà không giả sử rằng số lượng được dùng bởi khối xuất hiện trong cùng thanh ghi không có cách nào để đạt tới khối đó. Để tránh lỗi có thể xảy ra, giải thuật sinh mã đơn giản sẽ lưu giữ tất cả các giá trị khi đi qua ranh giới của khối cơ bản cũng như khi gọi chương trình con.

Ta có thể tạo ra mã phù hợp với câu lệnh ba địa chỉ *a := b + c* nếu ta tạo ra chỉ thị đơn ADD Rj, Ri với giá là 1. Kết quả a được đưa vào thanh ghi Ri chỉ nếu thanh ghi Ri chứa b, thanh ghi Rj chứa c, và b không được sử dụng nữa.

Nếu b ở trong Ri , c ở trong bộ nhớ , ta có thể tạo chỉ thị:

ADD c, Ri giá = 2

Hoặc nếu b ở trong thanh ghi Ri và giá trị của c được đưa từ bộ nhớ vào Rj sau đó thực hiện phép cộng hai thanh ghi Ri, Rj, ta có thể tạo các chỉ thị:

MOV c, R_j
ADD Rj , Ri giá = 3

Qua các trường hợp trên chúng ta thấy rằng có nhiều khả năng để tạo ra mã đích cho một lệnh ba địa chỉ. Tuy nhiên, việc lựa chọn khả năng nào lại tuỳ thuộc vào ngữ cảnh của mỗi thời điểm cần tạo mã.

1. Mô tả thanh ghi và địa chỉ

Giải thuật sinh mã đích dùng bộ mô tả (descriptor) để lưu giữ nội dung thanh ghi và địa chỉ của tên.

1. Bộ mô tả thanh ghi sẽ lưu giữ những gì tồn tại trong từng thanh ghi cũng như cho ta biết khi nào cần một thanh ghi mới. Ta giả sử rằng lúc đầu, bộ mô tả sẽ khởi động sao cho tất cả các thanh ghi đều rỗng. Khi sinh mã cho các khối cơ bản, mỗi thanh ghi sẽ giữ giá trị 0 hoặc các tên tại thời điểm thực hiện.
2. Bộ mô tả địa chỉ sẽ lưu giữ các vị trí nhớ nơi giá trị của tên có thể được tìm thấy tại thời điểm thực thi. Các vị trí đó có thể là thanh ghi, vị trí trên Stack, địa chỉ bộ nhớ. Tất cả các thông tin này được lưu chứa trong bảng danh biểu và sẽ được dùng để xác định phương pháp truy xuất tên.

2. Giải thuật sinh mã đích

Giải thuật sinh mã sẽ nhận vào chuỗi các lệnh ba địa chỉ của một khối cơ bản. Với mỗi lệnh ba địa chỉ dạng *x := y op z* ta thực hiện các bước sau:

1. Gọi hàm *getreg* để xác định vị trí L nơi lưu giữ kết quả của phép tính *y op z*. L thường là thanh ghi nhưng nó cũng có thể là một vị trí nhớ.

2. Xác định địa chỉ mô tả cho y để từ đó xác định y' một trong những vị trí hiện hành của y. Chúng ta ưu tiên chọn thanh ghi cho y' nếu cả thanh ghi và vị trí nhớ đang giữ giá trị của y. Nếu giá trị của y chưa có trong L, ta tạo ra chỉ thị:

MOV y', L để lưu bản sao của y vào L.

3. Tạo chỉ thị *op z', L* với z' là vị trí hiện hành của z. Ta ưu tiên chọn thanh ghi cho z' nếu giá trị của z được lưu giữ ở cả thanh ghi và bộ nhớ. Việc xác lập mô tả địa chỉ của x chỉ ra rằng x đang ở trong vị trí L. Nếu L là thanh ghi thì L là đang giữ trị của x và loại bỏ x ra khỏi tất cả các bộ mô tả thanh ghi khác.

4. Nếu giá trị hiện tại của y và/ hoặc z không còn được dùng nữa khi ra khỏi khối, và chúng đang ở trong thanh ghi thì sau khi ra khỏi khối ta phải xác lập mô tả thanh ghi để chỉ ra rằng các thanh ghi trên sẽ không giữ trị y và/hoặc z.

Nếu mã ba địa chỉ có phép toán một ngôi thì các bước thực hiện sinh mã đích cũng tương tự như trên.

Một trường hợp cần đặc biệt lưu ý là lệnh *x := y*. Nếu y ở trong thanh ghi, ta phải thay đổi thanh ghi và bộ mô tả địa chỉ, là giá trị của x được tìm thấy ở thanh ghi chứagiá trị của y. Nếu y không được dùng tiếp thì thanh ghi đó sẽ không còn lưu trị của y nữa. Nếu y ở trong bộ nhớ, ta dùng hàm getreg để tìm một thanh ghi tải giá trị của y và xác lập rằng thanh ghi đó là vị trí của x. Nếu ta thông báo rằng vị trí nhớ chứa giá trị của x là vị trí nhớ của y thì vấn đề trở nên phức tạp hơn vì ta không thể thay đổi giá trị của y nếu không tìm một chỗ khác để lưu giá trị của x trước đó.

3. Hàm getreg

Hàm getreg sẽ trả về vị trí nhớ L lưu giữ giá trị của x trong lệnh *x := y op z*. Sau đây là cách đơn giản dùng để cài đặt hàm:

1. Nếu y đang ở trong thanh ghi và y sẽ không được dùng nữa sau khi thực hiện *x := y op z* thì trả thanh ghi chứa y cho L và xác lập thông tin cho bộ mô tả địa chỉ của y rằng y không còn trong L.

2. Ngược lại, trả về một thanh ghi rỗng (nếu có).

3. Nếu không có thanh ghi rỗng và nếu x còn được dùng tiếp trong khối hoặc toán tử op cần thanh ghi, ta chọn một thanh ghi không rỗng R. Lưu giá trị của R vào vị trí nhớ M bằng chỉ thị *MOV R,M*. Nếu M chưa chứa giá trị nào, xác lập thông tin bộ mô tả địa chỉ cho M và trả về R. Nếu R giữ trị của một số biến, ta phải dùng chỉ thị MOV để lần lượt lưu giá trị cho từng biến.

4. Nếu x không được dùng nữa hoặc không có một thanh ghi phù hợp nào được tìm thấy, ta chọn vị trí nhớ của x như L.

Ví dụ 9.5 : Lệnh gán d := (a - b) + (a - c) + (a - c)

Có thể được chuyển sang chuỗi mã ba địa chỉ:

t := a - b
u := a - c

v := t + u
d := v + u

và d sẽ "sống" đến hết chương trình. Từ chuỗi lệnh ba địa chỉ nầy, giải thuật sinh mã vừa được trình bày sẽ tạo chuỗi mã đích với giả sử rằng: a, b, c luôn ở trong bộ nhớ và t, u, v là các biến tạm không có trong bộ nhớ .

Câu lệnh 3 địa chỉ	Mã đích	Giá	Bộ mô tả thanh ghi	Bộ mô tả địa chỉ
t := a - b	MOV a, R_0	2	Thanh ghi rỗng, R_0 chứa t	t ở trong R_0
	SUB b, R_0	2	R_0 chứa t	t ở trong R_0
u := a - c	MOV a, R_1	2		u ở rong R_1
	SUB c, R_1	2	R_1 chứa u	u ở trong R_1
v := t + u	ADD R_1, R_0	1	R_0 chứa v	v ở trong R_0
d := v + u	ADD R_1, R_0	1	R_1 chúa u	d ở rong R_0
	MOV R_0, d	2	R_0 chứa d	d ở trong bộ nhớ

Hình 9.9 - *Chuỗi mã đích*

Lần gọi đầu tiên của hàm *getreg* trả về R_0 như một vị trí để xác định t. Vì a không ở trong R_0 , ta tạo ra chỉ thỉ *MOV a,* R_0 và *SUB b,* R_0. Ta cập nhật lại bộ mô tả để chỉ ra rằng R_0 chứa t.

Việc sinh mã đích tiếp tục tiến hành theo cách nầy cho đến khi lệnh ba địa chỉ cuối cùng *d := v + u* được xử lý. Chú ý rằng R_0 là rỗng vì u không còn được dùng nữa. Sau đó ta tạo ra chỉ thị, cuối cùng của khối, MOV R_0, d để lưu biến "sống" d. Giá của chuỗi mã đích được sinh ra như ở trên là 12. Tuy nhiên, ta có thể giảm giá xuống còn 11 bằng cách thay chỉ thị *MOV a,* R_1 bằng *MOV* R_0, R_1 và xếp chỉ thị nầy sau chỉ thị thứ nhất.

4. Sinh mã cho loại lệnh khác

Các phép toán xác định chỉ số và con trỏ trong câu lệnh ba địa chỉ được thực hiện giống như các phép toán hai ngôi. Hình sau minh họa việc sinh mã đích cho các câu lệnh gán*: a := b[i], a[i] := b* và giả sử b được cấp phát tĩnh .

Câu lệnh 3 địa chỉ	(1) i trong thanh ghi R_i		(2) i trong bộ nhớ Mi		(3) i trên Stack	
	Mã	Giá	Mã	Giá	Mã	Giá
a:= b[i]	MOV b(R_i), R	2	MOV M_i, R MOV b(R), R	4	MOV S_i(A), R MOV b(R), R	4
a[i]:=b	MOV b, a(R_i)	3	MOV M_i , R MOV b, a (R)	5	MOV S_i(A), R MOV b, a (R)	5

Hình 9.10 - *Chuỗi mã đích cho phép gán chỉ mục*

Với mỗi câu lệnh ba địa chỉ trên ta có thể có nhiều đoạn mã đích khác nhau tuỳ thuộc vào i đang ở trong thanh ghi, hoặc trong vị trí nhớ M_i hoặc trên Stack tại vị trí S_i và con trỏ trong thanh ghi A chỉ tới mẩu tin hoạt động của i. Thanh ghi R là kết quả trả về khi hàm getreg được gọi. Đối với lệnh gán đầu tiên, ta đưa a vào trong R nếu a tiếp tục được dùng trong khối và có sẵn thanh ghi R. Trong câu lệnh thứ hai ta giả sử rằng a được cấp phát tĩnh.

Sau đây là chuỗi mã đích được sinh ra cho các lệnh gán con trỏ dạng *a := *p* và **p := a*. Vị trí nhớ p sẽ xác định chuỗi mã đích tương ứng.

Câu lệnh 3 địa chỉ	**p trong thanh ghi R_p**		**p trong bộ nhớ Mi**		**p trong Stack**	
	Mã	**Giá**	**Mã**	**Giá**	**Mã**	**Giá**
a:= *p	MOV *R_p, a	2	MOV M_p, R MOV *R, R	3	MOV S_p(A), R MOV *R, R	3
*p:= a	MOV a, *R_p	2	MOV M_p, R MOV a, *R	4	MOV a, R MOV R, *S_p(A)	4

Hình 9.11 - *Mã đích cho phép gán con trỏ*

Ba chuỗi mã đích tuỳ thuộc vào p ở trong thanh ghi R_p, hoặc p trong vị trí nhớ M_p, hoặc p ở trong Stack tại offset là S_p và con trỏ, trong thanh ghi A, trỏ tới mẩu tin hoạt động của p. Thanh ghi R là kết quả trả về khi hàm getreg được gọi. Trong câu lệnh gán thứ hai ta giả sử rằng a được cấp phát tĩnh.

5. Sinh mã cho lệnh điều kiện

Máy tính sẽ thực thi lệnh nhảy có điều kiện theo một trong hai cách sau:

1. Rẽ nhánh khi giá trị của thanh ghi được xác định trùng với một trong sáu điều kiện sau: âm, không, dương, không âm, khác không, không dương. Chẳng hạn, câu lệnh ba địa chỉ *if x < y goto z* có thể được thực hiện bằng cách lấy x trong thanh ghi R trừ y. Sau đó sẽ nhảy về z nếu giá trị trong thanh ghi R là âm.
2. Dùng tập các mã điều kiện để xác định giá trị trong thanh ghi R là âm, bằng không hay dương. Chỉ thị so sánh CMP sẽ kiểm tra mã điều kiện mà không cần biết trị tính toán cụ thể. Chẳng hạn, CMP x, y xác lập điều kiện dương nếu x > y,... Chỉ thị nhảy có điều kiện được thực hiện nếu điều kiện < , =, >, >=,<>, <= được xác lập. Ta dùng chỉ thị nhảy có điều kiện CJ <= z để nhảy đến z nếu mã điều kiện là âm hoặc bằng không.

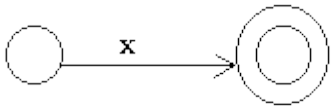
Chẳng hạn, lệnh điều kiện *if x < y goto z* được dịch sang mã máy như sau.

CMP x,y
CJ < z

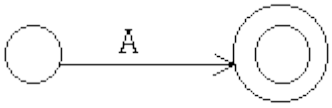
§Ó x©y dùng s¬ ®å dÞch cho ph¬ng ph¸p ph©n tÝch duyÖt lïi, tríc hÕt lo¹i bá ®Ö qui tr¸i, t¹o nh©n tè tr¸i cña v¨n ph¹m s¬ ®å dÞch theo nguyªn t¾c sau:

A1). Mçi ký hiÖu cha kÕt thóc t¬ng øng víi mét s¬ ®åtrong ®ã nh·n cho c¸c c¹nh lµ token hoÆc ký hiÖu cha kÕt thóc.

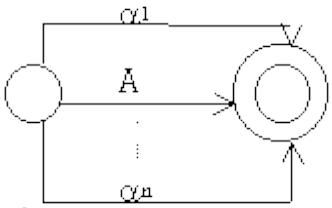
A2). Mỗi token tương ứng với việc đón nhận token đó và đọc token kế tiếp



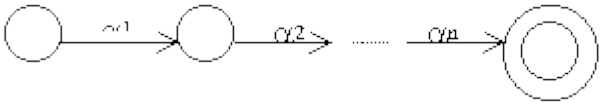
A3). Mỗi ký hiệu chưa kết thúc tương ứng với lời gọi thủ tục cho ký hiệu đó.



A4). Mỗi luật sinh có dạng $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ tương ứng với sơ đồ dịch



A5). Mỗi luật sinh dạng $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$ tương ứng với sơ đồ dịch



VÝ dô :XÐt v¨n ph¹m sinh biÓu thøc to¸n häc

E →E + T |T

E →T * E | F

F →E0 | id

Khö ®Ö qui tr¸i ta ®îc

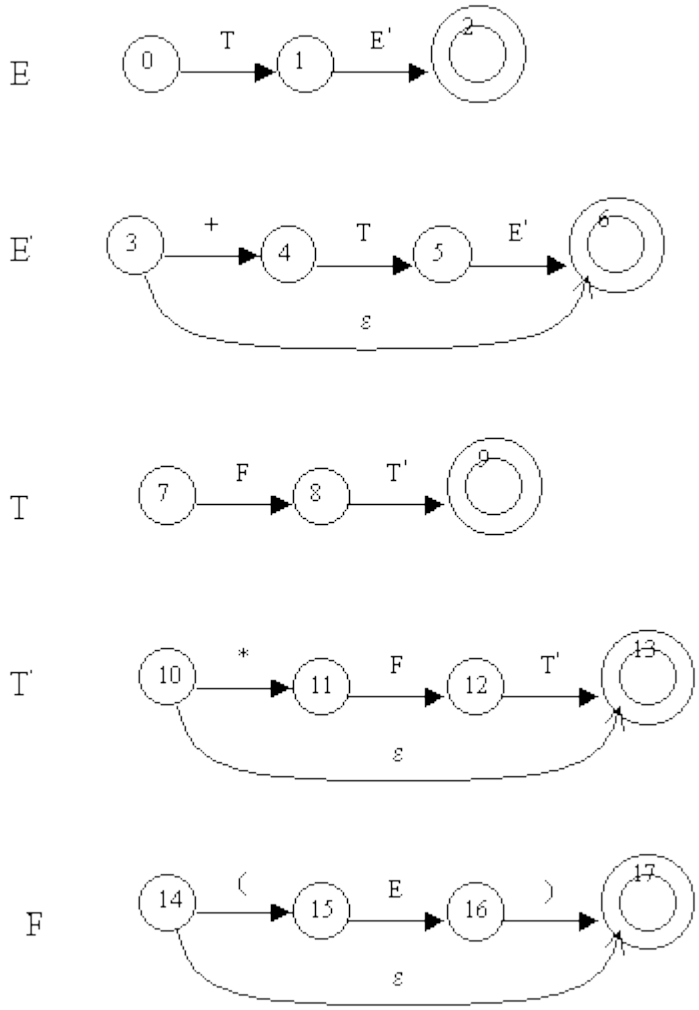
E →TE'

E' →+ TE' | ε

T →FT'

T' →* FT' | ε

F →(E) | id S¬ ®å dÞch t¬ng øng



Rót gän s¬ ®å b»ng c¸c thay thÕ t¬ng øng

CS 3240 Homework I

Scanning and Parsing

Let us consider the language of arithmetic expressions

The **alphabet** of this language is the set
`{+, -, *, /, (, ), x, y, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`

Note commas are not a part of the alphabet in the above set – they are only shown to separate elements of the set. That is, **strings** in this language can be composed only by using one or more of the following
`+ - * / ( ) x y 0 1 2 3 4 5 6 7 8 9`

The **tokens** in this language are of the following classes

```
MOPER     : * /
AOPER     : + -
CONS      : Strings made of 0 through 9
VAR       : x y
OPARAN    : (
CPARAN    : )
```

Consider a compiler that scans and parses the language of arithmetic expressions

Question 1: As you scan the following expression from left to right, list the tokens and the token class identified by the **scanner** for each of the arithmetic expressions below. **Identify, explain** and clearly **mark** the errors if any (30 points)

```
a.   ( x * ( y + 100 ) + y - ( x + y - 320 ) )
b.   ( y + 100 * x + ( 2 + x^3 ) / y )
c.   x * ) 4 + / 100 - y
d.   y * ( ( x + 100
e.   (20 + x * 4 / 30y3 )
```

The **grammar** for the language of arithmetic expressions is as follows

```
<EXPR>    →   <TERM> AOPER <TERM>
<EXPR>    →   <TERM>
<TERM>    →   <FAC> MOPER <FAC>
<TERM>    →   <FAC>
<FAC>     →   OPARAN <EXPR> CPARAN
<FAC>     →   VAR
<FAC>     →   CONS
```

Question 2: What are the **terminals** and **non-terminals** in this grammar? (10 points)

Question 3: For each of the expressions below, scan it from left to right; list the tokens returned by the scanner and the **rules** used by the parser (showing appropriate expansions of the non-

terminals) for matching. **Identify, explain** and clearly **mark** the errors if any (40 points)

a.	a.	`( x + y )`
b.	b.	`( y * - x ) + 10`
c.	c.	`( x * ( y + 10 ) )`
d.	d.	`( x + y ) * ( y + z )`
e.	e.	`( x + ( y - ( 2 ) )`

Question 4: You are asked the count the number of constants (**CONS**), variables (**VAR**) and **MOPER** in an expression. Insert action symbols in the grammar described before Question 2, explain what semantic actions they trigger and what each semantic action does. (20 points)

Regular Expressions

Question 1: Consider the concept of "closure". A set S is said to be closed under a (binary) operation ⊕ if and only if applying the operation to two elements in the set results in another element in the set. For example, consider the set of natural numbers **N** and the "+" (addition) operation. If we add any two natural numbers, we get a natural number. Formally x, y are elements of **N** implies x + y is an element of **N**. **State true or false and explain why**

a. Only infinite sets (sets with infinite number of elements, like the set of natural numbers) can be closed

b. Infinite sets are closed under all operations

c. The set **`[a-z]*`** is closed under concatenation operation

Question 2:

For each of the regular expressions below, state if they describe the same set of strings (state if they are equivalent). If they are equivalent, what is the string they describe?

1.	**`[a-z][a-z]*`**	and	**`[a-z]+`**
2.	**`[a-z0-9]+`**	and	**`[a-z]+[0-9]+`**
3.	**`[ab]?[12]?`**	and	**`a1|b1|a2|b2`**
4.	**`[ab12]+`**	and	**`a|b|1|2|[ab12]*`**
5.	**`[-az]*`**	and	**`[a-z]*`**
6.	**`[abc]+`**	and	**`[cba]+`**

7. **`[a-j][k-z]`** and **`[a-z]`**

<u>Question 3</u>:

For each of the strings described below, write a regular expression that describes them and draw a finite automaton that accepts them.

1. 1. The string of zero or more **`a`** followed by three **`b`** followed zero or more **`c`**

2. 2. The string of zero or more **`a, b`** and **`c`** but every **`a`** is followed by two or more **`b`**

3. 3. All strings of digits that represent even numbers

4. 4. All strings of **`a`**'s and **`b`**'s that contain no three consecutive **`b`**'s.

5. 5. All strings that can be made from **`{0, 1}`** except the strings **`11`** and **`111`**

Question 1: Pumping Lemma and Regular Languages

You can use the pumping lemma and the closure of the class of regular languages under union, intersection and complement to answer the following question. Proofs should be rigorous. Note that for each of the questions below, you may or may not have to use the pumping lemma.

Note that the notation 0_m means “0 repeated *m* times”. So the language of strings of the form 0_m such that *m* iÝ 0 would contain strings like the null string 0, 00, 000, … (this is [0]*. Whereas the language of strings of the form 0_m such that *m* iÝ 1 would be [0]+)

a. Is the language of strings of the form $0_m 1_n 0_m$ such that *m, n* iÝ *0* regular? If it is regular, **prove** that it is regular. If it is not regular, **prove** that is not regular. Note that, a rigorous proof is needed. General reasoning or explanations that are not rigorous will not get full credit. (15 points)

b. Consider a language whose alphabet is from the set {a, b}. Is the language of *palindromes* over this alphabet regular? If it is regular, **prove** that it is regular. If it is not regular, **prove** that is not regular. Note that, a rigorous proof is needed. General reasoning or explanations that are not rigorous will not get full credit. (15 points)

Hint: A palindrome is a word such that when read backwards, is the same word. For example the word “mom” when read left to right is the same as it is when it is read right to left. In general, the first half, when reversed, yields the second half. If the length of the string is odd, the middle character is left as it is. For example, consider the word “redivider”. Reversing “redi” yields “ider” and “v” is left as it is. For strings with alphabet {a, b}, “aaabaaa” is a palindrome but “abaaa” is not.

c. A language, whose alphabet is {a, b}, such that the strings of the language contain equal number of “ab” and “ba”. Note that “aba” is part of the language, because the first letter and the second letter form “ab” and the second and third form “ba”. Is this language regular? If it is regular, **prove** that it is regular. If it is not regular, **prove** that is not

regular. Note that, a rigorous proof is needed. General reasoning or explanations that are
not rigorous will not get full credit. (15 points)
d. The class of regular languages is closed under union. That is of A is a regular language
and B is a regular language, then C is a regular language, where C = A . B. Note that B
. C. (B is a subset of C). Let D be some subset of C (that is, D . C). In general, is D
regular? If it is regular, **prove** that it is regular. If it is not regular, **prove** that is not
regular. Note that, a rigorous proof is needed. General reasoning or explanations that are
not rigorous will not get full credit. (15 points)
Question 2:
Consider the language described by the regular expression a+b*a, the set of all strings
that has one or more a's followed by zero or more b's and ending in a single a.
a. Construct a NFA which recognizes this language. Note that you need to construct a
primitive NFA using the constructions describe in class. (10 points)
b. Convert the above NFA to a DFA using . closure. Clearly indicate the steps of .
closure. (20 points)
c. Convert the above DFA to an optimized DFA (10 points)

HomeWork

1. Work on the homework individually. Do not collaborate or copy from others
2. The homework is due on Tuesday, April 24 In Class. No late submissions will be entertained
3. Do not email your answers to either the Professor or the TA. Emailed answers will not be considered for evaluation

Question 1. (50 Points)

Consider the following grammar. Construct LR(0) items, DFA for this grammar showing LR(0) shiftreduce table. Is this grammar LR(0)? Indicate all possible shift-reduce as well as reduce-reduce conflicts. Using the concept of look-ahead, generate SLR(1) table – which LR(0) conflicts get eliminated? Using the input (ID + ID) * ID show the SLR(1) parse - show the stack states and shifts and reductions as shown in the examples in the Louden book.

Grammar:

E' -> E
E -> E + T
E -> T
T -> T * ID
T -> ID
T -> (E)

Question 2. (50 Points)

Construct a pushdown automaton for the following language:

L = { $a_i b_j c_k$ | i, j, k >= 0, either i = j or j = k}

Practice

Q #1. Design a Turing machine for recognizing the language (please give a formal
description including tape alphabet, full state transition diagram identifying the
acceptance and rejection states if any)

$L = \{a_n b_n c_n \mid n >= 0\}$

L = { w | w contains twice as many 0's as 1's, w is made from {0,1}* }

Q #2. Design a Turing machine to perform multiplication of two natural numbers
represented as the number of zeroes. For example, number five is represented as 00000

Hint: Use repeated addition

Q #3 Design LR(0) items, their DFA and SLR(1) parse table for the following grammar
showing the parse for the following input : ((a), a, (a, a)) Also show the parse tree
obtained. Is this a LR(0) grammar? If not show the conflicts and show how you can
resolve them through SLR(1) construction

Grammar :

E -> (L)| a

L -> L, E| E

Q #4 Design Context free grammars for the following languages (alphabet is {0,1})

a. {w | w starts and ends with the same symbol (either 0 or 1, which is the alphabet)}

b. $\{w \mid w = w_r$ ie, w is a palindrome}

c. $\{a_i b_j c_k \mid i = j$ or $j = k, i, j, k >= 0\}$

Q #5 Design pushdown automata (PDA) for the following language:

{w | w has odd length and the middle character is 0}

Q #6 Show first, follow and predict sets for the following grammar after removing left
recursion and left factoring:

E -> E + T

E -> T

T -> T * P

T -> P

P -> (E)

P -> ID

Q # 7 Using the pumping lemma show that the following languages are not regular:

$\{0_m 1_n \mid m$ not equal to n}

$\{0_{2n}$

| n >= 0}

Q #8 Design NFA, DFA and minimize the DFA for the regular expression:

$0^*1^*0^*0$

Test 1

Question 1: DFAs (Choose any three questions out of five: 30 points)

Devise DFAs for:

1. All strings that start with 1 must end with a 0 and those which start with 0 must end with 1 (alphabet of this language is {0,1}), no null string
2. All strings from the alphabet {a, b} which contain an odd number of a's and even (but non-zero) number of b's
3. All strings that must have 0110 as the substring (alphabet {0,1})
4. All strings which have a length greater than or equal to 3 and ending on b or two consecutive a's
5. Strings that do not contain 3 consecutive a's

Question 2: Regular expressions (Choose any three questions out of five: 30 points)

Write regular expressions for:

1. Expressions that enumerate all positive integers (including 0) upto 100000 but without any leading zeroes
2. Strings made from {a, b} that start and end on the same letter (ie, strings starting with a end on a and those starting with b end on b)
3. Floats using decimal point representation with integer and fractional parts – no leading or trailing zeros and precision upto 4 places after decimal
4. Identifiers that start with a digit or lowercase letter following which one can optionally have one or more of digits or letters or underscores. Identifiers can not end on an underscore (consecutive underscores ok though)
5. Positive integers no leading zeros in which all 2's should occur only after 3's and all 1's should occur only after 2's (ie, no 2 should occur before a 3 or no 1 should occur before a 2).

Question 3: **Regular Expression** . **NFA** . **DFA (30 points)**

Convert the following regular expression into a NFA and convert the NFA to DFA
showing the key steps (such as computing å-closures of sets of states etc.) : b[ab]∗ Show
all possible NFA transitions (using parallel tree) for the string babba and verify the state
transitions in corresponding DFA

Question 4: State True or False (10 points)

a. Consider a language S=(a|b)*. Consider a Regular Language L, whose alphabet is
from the set .= {a, b}. Let M be a DFA that Recognizes L. Let M' be a DFA obtained from M by changing all accepting states of the M into non-accepting states, and by changing all non-accepting states of M to accepting states. M' recognizes the complement of language L given by S – L

b. For every NFA and its equivalent DFA, the number of states in equivalent DFA
must be at least equal to the number of states in the NFA.

c. Consider languages L and L' such that L . L'. Let M be a DFA that recognizes L

and M’ be DFA that recognizes L’ then the number of states in M’ must be equal
to or greater than those in M.
d. Consider languages L and L’ such that L . L’. Let M be a DFA that recognizes L
and M’ be DFA that recognizes L’ then the number of states in M’ must be lesser
than or equal to those in M.
e. For every regular expression there can exist more than one DFA that recognizes
the language described by the regular expression.
.

Tesst 2

Project

Notes:
1. This project has two phases. Phase 1 is due by April 14th by 5pm. Phase 2 is due by April 28th
by 5pm.
2. There will be no extensions for either phases
3. You will work in groups of three
4. Each group should submit a report and source code for each phase. If multiple source files, they
must be tarred along with the makefile
5. You can program in C, C++ or Java. Do not use tools (like lex and yacc) or the standard
template library
6. Code should be properly documented with meaningful variable and function names. Short
elegant code will get bonus points.
7. You will find the course slides on DFA/NFA/scanner/recursive descent parser useful.
8. Each phase of the project is worth 100 points. The bonus section is worth 50 points.

Phase 1:

Objective: To write a scanner and parser which can construct and execute an NFA for any regular
expression.
Consider the language of regular expressions. The alphabet of this language is the set `{a, b, *,
+, (,), ., |}` (commas and spaces are not part of the language). Using this alphabet one can
write any regular expression. Our goal in this project is to be able to read any regular expression
described by the following grammar and construct primitive NFAs and join them together to form a
NFA that will recognize strings described by the regular expression. We will do this step by step by
developing answers to the following questions. The production rules for this language are given by

R . R*
R . R+
R . (R)
R . (R | R)
R . R.R
R . a
R . b

Question 1: Rewrite the grammar to remove left recursion.
Question 2: Identify the tokens of this language and write a scanner program which can scan this
language and return tokens .

Question 3: Write a *recursive descent* parser which can parse this language (based on the modified
grammar which removed left recursion) and yield a parse tree. Note that this grammar has implicit
precedence. That is for a regular expression, a.b* the “*” operates on “b” and not a.b as a whole. This
is true unless it is bracketed. In, (a.b)* on the other hand, the “*” operates on (a.b) When you build a
parse tree you must take care of such precedences
Question 4: Now you need to write a program which can construct a NFAs based on the parse tree
based on primitive NFAs. As discussed in class, primitive NFAs should be joined together to form
NFA for the complete regular expression. This final NFA will be represented as an adjacency matrix
described below. Thus the output of this program should be an *adjacency matrix.*
Adjacency matrix: Any NFA is a *directed graph*. A directed graph G consists of a set of *nodes* (in our
case states) and *directed edges* (in our case, transitions). For example, in the graph below, A,B,C are
nodes and 1,2,3 are edges
A
B
C
1 2
3
Any directed graph can be represented by an adjacency matrix. For example, the matrix below
represents the graph. Since edge “1” connects A to B, there is a “1” in the row corresponding to “A”
and the column corresponding to “B”.
A B C
A 1 3
B 2
C
Similarly an NFA can be represented by an adjacency matrix. Note that more than one element can be
present in a cell. For example, in the NFA if the edge from A to B is labeled a,b then you would have
both “a” and “b” in the corresponding cell.
Question 5: Given such an adjacency matrix of an NFA and given an input string consisting of a’s and
b’s write a program to *simulate* the NFA and output if the string is accepted or rejected. Note : NFAs
can progress on multiple paths and you should simulate this effect – if one of the paths results in accept
state then the input string is accepted by NFA.

Phase 2: To write a program which will construct a DFA from any NFA. You will use adjacency
matrix as the representation and use epsilon closures to generate DFA. Finally write a program to
simulate the DFA.

Bonus: Given an adjacency matrix for a DFA, write a program to produce minimal DFA by state
merging.