CHƯƠNG VIII
SINH MÃ TRUNG GIAN

Nội dung chính:
Thay vì một chương trình nguồn được dịch trực tiếp sang mã đích, nó nên được dịch sang dạng *mã trung gian* bởi kỳ trước trước khi được tiếp tục dịch sang mã đích bởi kỳ sau vì một số tiện ích: Thuận tiện khi muốn thay đổi cách biểu diễn chương trình đích; Giảm thời gian thực thi chương trình đích vì mã trung gian có thể được tối ưu. Chương này giới thiệu *các dạng biểu diễn trung gian* đặc biệt là dạng *mã ba địa chỉ*. Phần lớn nội dung của chương tập trung vào trình bày cách tạo ra một *bộ sinh mã trung gian* đơn giản dạng mã 3 đại chỉ. Bộ sinh mã này dùng phương thức trực tiếp cú pháp để dịch các khai báo, câu lệnh gán, các lệnh điều khiển sang mã ba địa chỉ.

Mục tiêu cần đạt:
Sau khi học xong chương này, sinh viên phải nắm được cách tạo ra một bộ sinh mã trung gian cho một ngôn ngữ lập trình đơn giản (chỉ chứa một số loại khai báo, lệnh điều khiển và câu lệnh gán) từ đó có thể mở rộng để cài đặt bộ sinh mã cho những ngôn ngữ phức tạp hơn.

Tài liệu tham khảo:

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

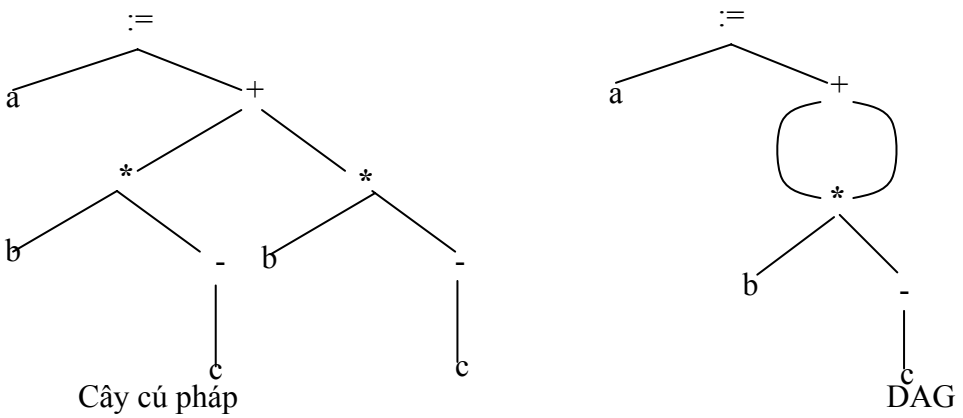
I. NGÔN NGỮ TRUNG GIAN

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

1. Biểu diễn đồ thị

Cây cú pháp mô tả cấu trúc phân cấp tự nhiên của chương trình nguồn. DAG cho ta cùng lượng thông tin nhưng bằng cách biểu diễn ngắn gọn hơn trong đó các biểu thức con không được biểu diễn lặp lại.

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



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

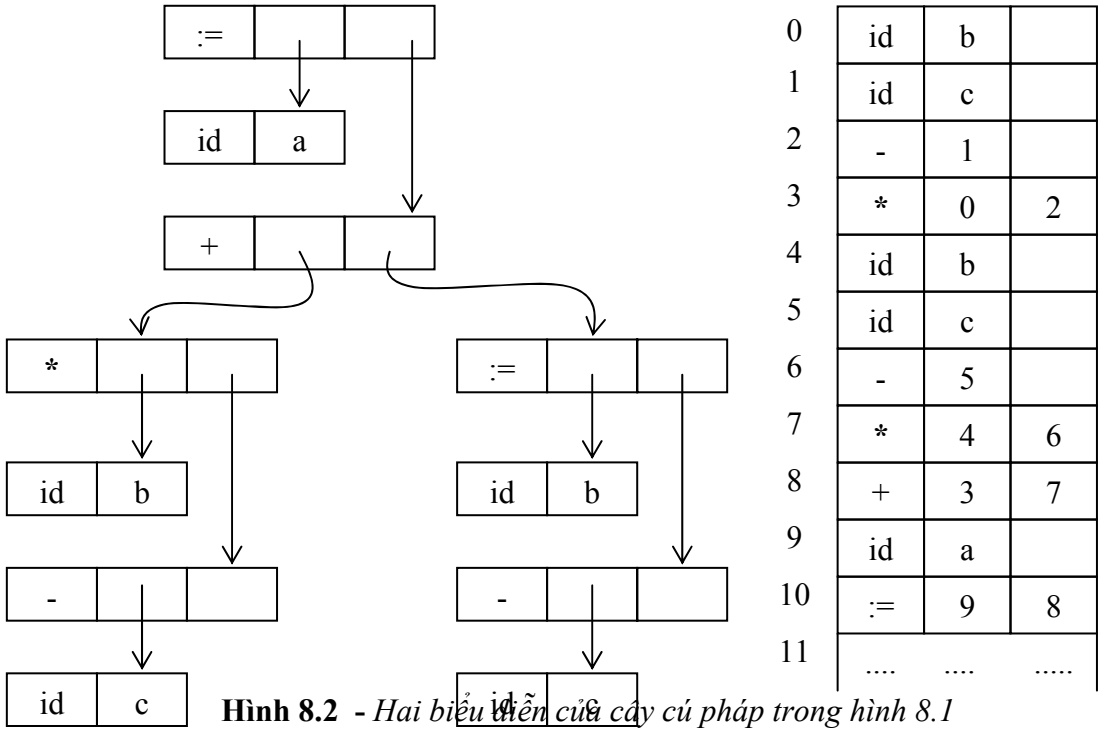
Ký pháp hậu tố là một biểu diễn tuyến tính của cây cú pháp. Nó là một danh sách các nút của cây, trong đó một nút xuất hiện ngay sau con của nó.

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

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

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

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



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

2. Mã 3 địa chỉ

Mã lệnh 3 địa chỉ là một chuỗi các lệnh có dạng tổng quát là **x :=y op z**. Trong đó x,y,z là tên, hằng hoặc dữ liệu tạm sinh ra trong khi dịch, op là một toán tử số học hoặc logic.

Chú ý rằng không được có quá một toán tử ở vế phải của mỗi lệnh. Do đó biểu thức x + y * z phải được dịch thành chuỗi :

t1 := y * z

t2 := x + t1

Trong đó t1, t2 là những tên tạm sinh ra trong khi dịch.

Mã lệnh 3 địa chỉ là một biểu diễn tuyến tính của cây cú pháp hoặc DAG, trong đó các tên tường minh biểu diễn cho các nút trong trên đồ thị.

t1 := - c
*t2 := b * t1*
t3 := - c
*t4 := b * t3*

t1 := -c
*t2 := b * t1*
t3 := t2 + t2
a := t3

t5 := t2 + t4

a := t5

Mã lệnh 3 địa chỉ của cây cú pháp Mã lệnh 3 địa chỉ của DAG

Hình 8.3 - *Mã lệnh 3 địa chỉ tương ứng với cây cú pháp và DAG trong hình 8.1*

3. Các mã lệnh 3 địa chỉ phổ biến

1. Lệnh gán dạng **x := y op z**, trong đó op là toán tử 2 ngôi số học hoặc logic.
2. Lệnh gán dạng **x := op y**, trong đó op là toán tử một ngôi. Chẳng hạn, phép lấy số đối, toán tử NOT, các toán tử SHIFT, các toán tử chuyển đổi.
3. Lệnh COPY dạng **x :=y**, trong đó giá trị của y được gán cho x.
4. Lệnh nhảy không điều kiện **goto L**. Lệnh 3 địa chỉ có nhãn L là lệnh tiếp theo thực hiện.
5. Các lệnh nhảy có điều kiện như **if x relop y goto L**. Lệnh này áp dụng toán tử quan hệ relop (<, =, >=, ...) vào x và y. Nếu x và y thỏa quan hệ relop thì lệnh nhảy với nhãn L sẽ được thực hiện, ngược lại lệnh đứng sau IF x relop y goto L sẽ được thực hiện.
6. **param x và call p, n** cho lời gọi chương trình con và return y. Trong đó, y biểu diễn giá trị trả về có thể lựa chọn. Cách sử dụng phổ biến là một chuỗi lệnh 3 địa chỉ.

   param x1

   param x2

   .. .. ..

   param xn

   call p, n

   được sinh ra như là một phần của lời gọi chương trình con p (x1,x2,.. .., xn).
7. Lệnh gán dạng **x := y[i] và x[i] := y**. Lệnh đầu lấy giá trị của vị trí nhớ của y được xác định bởi giá trị ô nhớ i gán cho x. Lệnh thứ 2 lấy giá trị của y gán cho ô nhớ x được xác định bởi i.
8. Lệnh gán địa chỉ và con trỏ dạng **x :=&y , x := *y và *x := y**. Trong đó, x := &y đặt giá trị của x bởi vị trí của y. Câu lệnh x := *y với y là một con trỏ mà r_value của nó là một vị trí, r_value của x đặt bằng nội dung của vị trí này. Cuối cùng *x := y đặt r_value của đối tượng được trỏ bởi x bằng r_value của y.

4. Dịch trực tiếp cú pháp thành mã lệnh 3 địa chỉ

Ví dụ 8.2: Định nghĩa S _ thuộc tính sinh mã lệnh địa chỉ cho lệnh gán:

Luật sinh	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.place := newtemp;
	E.code := E_1.code|| gen (E.place ':=' 'uminus' E_1.place)
E→ - E_1	*E.place := newtemp;*
	E.code := E_1.code
E→ (E_1)	*E.place := id.place;*
	E.code := ''
E→ id	

Hình 8.4 - *Định nghĩa trực tiếp cú pháp sinh mã lệnh ba địa chỉ cho lệnh gán*

Với chuỗi nhập a = b * - c + b * - c, nó sinh ra mã lệnh 3 địa chỉ

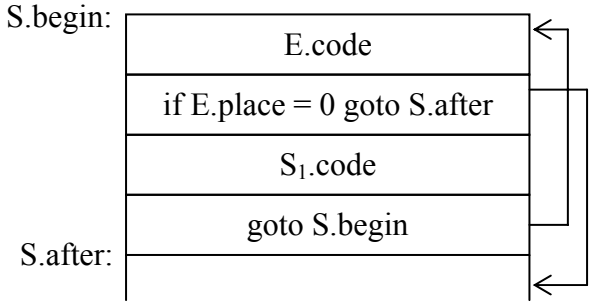
t1 := -c
*t2 := b * t1*
t3 := - c
*t4 := b * t3*
t5 := t2 + t4
a := t5

thuộc tính tổng hợp S.code biểu diễn mã 3 địa chỉ cho lệnh gán S. Ký hiệu chưa kết thúc E có 2 thuộc tính E.place là giá trị của E và E.code là chuỗi lệnh 3 địa chỉ để đánh giá E

Khi mã lệnh 3 địa chỉ được sinh, tên tạm được tạo ra cho mỗi nút trong trên cây cú pháp.

Giá trị của ký hiệu chưa kết thúc E trong luật sinh E → E1 + E2 được tính vào trong tên tạm t. Nói chung mã lệnh 3 địa chỉ cho lệnh gán id := E bao gồm mã lệnh cho việc đánh giá E vào trong biến tạm t, sau đó là một lệnh gán id.place := t.

Hàm newtemp trả về một chuỗi các tên t1, t2, ... , tn tương ứng các lời gọi liên tiếp. Gen (x ':=' y '+' z) để biểu diễn lệnh 3 địa chỉ x *:=y+z*



Luật sinh	**Luật ngữ nghĩa**
S→ while E do S_1	*S.begin := newlabel;*
	S.after := newlabel; S.code := gen(S.begin ':') || E.code ||
	gen('if' E.place '=' 0 'goto' S.after) ||
	S_1.code || *gen('*goto*' S.begin)* || *gen(S.after ':')*

Hình 8.5 - *Định nghĩa trực tiếp cú pháp sinh mã lệnh ba địa chỉ cho câu lệnh while*

Lệnh S → while E do S1 được sinh ra bằng cách dùng các thuộc tính S.begin và S.after để đánh dấu lệnh đầu tiên trong đoạn mã lệnh của E và lệnh sau đoạn mã lệnh của S.

Hàm newlabel trả về một nhãn mới tại mỗi lần được gọi

5. Cài đặt lệnh 3 địa chỉ

Lệnh 3 địa chỉ là một dạng trừu tượng của mã lệnh trung gian. Trong chương trình dịch, các mã lệnh này có thể cài đặt bằng một mẩu tin với các trường cho toán tử và toán hạng. Có 3 cách biểu diễn là bộ tứ, bộ tam và bộ tam gián tiếp.

- **Bộ tứ**

  Bộ tứ (quadruples) là một cấu trúc mẩu tin có 4 trường ta gọi là op, arg1, arg2 và result. Trường op chứa toán tử. Lệnh 3 địa chỉ ***x := y op z*** được biểu diễn bằng cách thay thế y bởi arg1, z bởi arg2 và x bởi result. Các lệnh với toán tử một ngôi như x := -y hay x := y thì không sử dụng arg2. Các toán tử như param không sử dụng cả arg2 lẫn result. Các lệnh nhảy có điều kiện và không điều kiện đặt nhãn đích trong result.

  Nội dung các trường arg1, arg2 và result trỏ tới ô trong bảng ký hiệu đối với các tên biểu diễn bởi các trường này. Nếu vậy thì các tên tạm phải được đưa vào bảng ký hiệu khi chúng được tạo ra.

  Ví dụ 8.3: Bộ tứ cho lệnh a := b * -c + b * -c

	op	**arg1**	**arg2**	**result**
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Hình 8.6 - *Biểu diễn bộ tứ cho các lệnh ba địa chỉ*

- **Bộ tam**

  Để tránh phải lưu tên tạm trong bảng ký hiệu; chúng ta có thể tham khảo tới giá trị tạm bằng vị trí của lệnh tính ra nó. Để làm điều đó ta sử dụng bộ tam (triples) là một mẩu tin có 3 trường op, arg1 và arg2. Trong đó, arg1 và arg2 trỏ tới bảng ký hiệu (đối với tên hoặc hằng do người lập trình định nghĩa) hoặc trỏ tới một phần tử trong bộ tam (đối với giá trị tạm)

	op	**arg1**	**arg2**
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Hình 8.7 - *Biểu diễn bộ tam cho các lệnh ba địa chỉ*

Các lệnh như x[i]:=y và x:=y[i] sử dụng 2 ô trong cấu trúc bộ tam.

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

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

Hình 8.8 - *Biểu diễn bộ tam cho x[i]:=y và x:=y[i]*

- **Bộ tam gián tiếp**

Một cách biểu diễn khác của bộ tam là thay vì liệt kê các bộ tam trực tiếp ta sử dụng một danh sách các con trỏ các bộ tam.

	statements
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Hình 8.9 - *Biểu diễn bộ tam gián tiếp cho các lệnh ba địa chỉ*

II. KHAI BÁO

1. Khai báo trong chương trình con

Các tên cục bộ trong chương trình con được truy xuất đến thông qua địa chỉ tương đối của nó. Gọi là offset.

Ví dụ 8.4: Xét lược đồ dịch cho việc khai báo biến

P → {*offset:= 0* } D

D → D ; D

D → id : T {*enter(id.name, T,type, offset); offset := offset + T.width* }

T → integer { *T.type:= integer; T.width := 4* }

T → real { *T.type:= real ; T.width := 8* }

T → array[num] of T1 { *T.type:= array(num.val, T1.type);*
*T.width := num.val * T1.width* }

T→ ↑ T1 { *T.type := pointer(T1.type) ; T.width := 4* }

Hình 8.10 - *Xác định kiểu và địa chỉ tương đối của các tên được khai báo*

Trong ví dụ trên, ký hiệu chưa kết thúc P sinh ra một chuỗi các khai báo dạng id:T.

Trước khi khai báo đầu tiên được xét thì offset = 0. Khi mỗi khai báo được tìm thấy tên và giá trị của offset hiện tại được đưa vào trong bảng ký hiệu, sau đó offset được tăng lên một khoảng bằng kích thước của đối tượng dữ liệu được cho bởi tên đó.

Thủ tục **enter(name, type, offset)** tạo một ô trong bảng ký hiệu với tên, kiểu và địa chỉ tương đối của vùng dữ liệu của nó. Ta sử dụng các thuộc tính tổng hợp type và width để chỉ ra kiểu và kích thước (số đơn vị nhớ) của kiểu đó.

Chú ý rằng lược đồ dịch P → {offset := 0} D có thể được viết lại bằng cách thay thế hành vi {offset := 0} bởi một ký hiệu chưa kết thúc M để được

$P \rightarrow M\ D$

$M \rightarrow \varepsilon\ \{offset := 0\}$

Tất cả các hành vi đều nằm cuối vế phải.

2. Lưu trữ thông tin về tầm

Trong một ngôn ngữ mà chương trình con được phép khai báo lồng nhau. Khi một chương trình con được tìm thấy thì quá trình khai báo của chương trình con bao bị tạm dừng.

Văn phạm cho sự khai báo đó là;

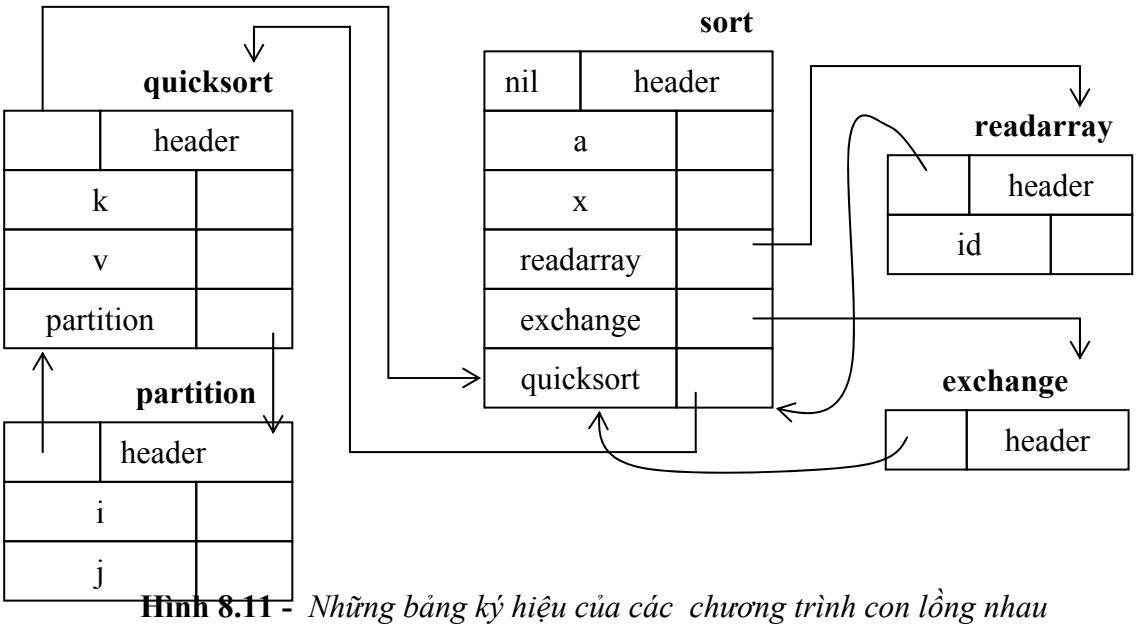
$P \rightarrow D$

$D \rightarrow D\ ;\ D \mid id:\ T \mid proc\ id\ ;\ D;\ S$

Để đơn giản chúng ta tạo ra một bảng ký hiệu riêng cho mỗi chương trình con.

Khi một khai báo chương trình con $D \rightarrow$ proc id D1 ; S được tạo ra và các khai báo trong D1 được lưu trữ trong bảng ký hiệu mới.

Ví dụ 8.5: Chương trình Sort có bốn chương trình con lồng nhau readarray, exchange, quicksort và partition. Ta có năm bảng ký hiệu tương ứng.



Hình 8.11 - *Những bảng ký hiệu của các chương trình con lồng nhau*

Luật ngữ nghĩa được xác định bởi các thao tác sau

1. *mktable (previous)*: Tạo một bảng ký hiệu mới và con trỏ tới bảng đó. Tham số previous là một con trỏ trỏ tới bảng ký hiệu của chương trình con bao. Con trỏ previous được lưu trong header của bảng ký hiệu mới. Trong header còn có thể có các thông tin khác như độ sâu lồng của chương trình con.
2. *enter (table, name, type, offset)*: Tạo một ô mới trong bảng ký hiệu được trỏ bởi table.
3. *addwidth (table, width)*: Ghi kích thước tích lũy của tất cả các ô trong bảng vào trong header kết hợp với bảng đó.

4. *enterproc (table, name, newtable)*: Tạo một ô mới cho tên chương trình con vào trong bảng được trỏ bởi table. newtable trỏ tới bảng ký hiệu của chương trình con này.

Ta có lược đồ dịch

$P \rightarrow M\ D$	*{ addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset) }*
$M \rightarrow \varepsilon$	*{ t:=mktable(nil); push(t, tblptr) ; push(0,offset) }*
$D \rightarrow D_1\ ;\ D_2$	
$D \rightarrow$ proc id ; $N\ D_1$; S	*{t:= top(tblptr); addwidth(t, top(offset)); pop(tblptr; pop(offset); enterproc(top(tblptr), id_name,t) }*
$D \rightarrow$ id : T	*{enter(top(tblptr), id_name, T.type, top(offset)); top(offset):= top(offset) + T.width }*
$N \rightarrow \varepsilon$	*{t:=mktable(top(tblptr)); push(t, tblptr); push(0,offset) }*

Hình 8.12 - *Xử lý các khai báo trong những chương trình con lồng nhau*

Ta dùng Stack **tblptr** để giữ các con trỏ bảng ký hiệu.

Chẳng hạn, khi các khai báo của partition được khảo sát thì trong tblptr chứa các con trỏ của các bảng của sort, quicksoft và partition. Con trỏ của bảng hiện hành nằm trên đỉnh Stack.

offset là một Stack khác để lưu trữ offset. Chú ý rằng với lược đồ *A→ BC {action A}* thì tất cả các hành vi của các cây con B và C được thực hiện trước A.

Do đó hành vi kết hợp với M được thực hiện trước. Nó không tạo ra bảng ký hiệu cho tầng ngoài cùng (chương trình sort) bằng cách dùng **mktable(nil)**, con trỏ tới bảng này được đưa vào trong Stack tblptr đồng thời được đưa vào Stack offset.

Ký hiệu chưa kết thúc N đóng vai trò tương tự như M khi một khai báo chương trình con xuất hiện. Nó dùng **mktable(top(tblptr))** để tạo ra một bảng mới. Tham số **top(tblptr**) cho giá trị con trỏ tới bảng lại được push vào đỉnh Stack tblptr và 0 được push vào Stack offset.

Với mỗi khai báo biến id:T; một ô mới được tạo ra trong bảng ký hiệu hiện hành. Giá trị trong top(offset) được tăng lên bởi T.width.

Khi hành vi vế phải P → proc id ; N D1 ; S diễn ra, kích thước của tất cả các đối tượng dữ liệu khai báo trong D1 sẽ nằm trên đỉnh Stack offsert. Nó được lưu trữ bằng cách dùng addwidth, các Stack tblptr và offset bị pop và chúng ta trở về để thao tác trên các khai báo của chương trình con

3. Xử lý đối với mẩu tin

Khai báo một mẩu tin được cho bởi luật sinh

$T \rightarrow$ **record** D **end**

Luật dịch tương ứng

$T \rightarrow$ **record** L D **end**	*{ T.type := record(top(tblptr)); T.width := top(offset); pop(tblptr) ; pop(offset) }*
$L \rightarrow \varepsilon$	*{ t:= mktable(nil); push(t,tblptr) ; push(0,offset) }*

Hình 8.13 - *Cài đặt bảng ký hiệu cho các tên trường trong mẩu tin*

Sau khi từ khóa record được tìm thấy thì hành vi kết hợp với L tạo một bảng ký hiệu mới cho các tên trường. Các hành vi của D → id : T đưa thông tin về tên trường id vào trong bảng ký hiệu cho mẩu tin.

III. LỆNH GÁN

1. Tên trong bảng ký hiệu

Xét lược đồ dịch để sinh ra mã lệnh 3 địa chỉ cho lệnh gán:

S→ id := E {*p:=lookup(id.name);*
if p <> nil then emit(p ':=' E.place) else error }

E→ E_1 + E_2 { *E.place := newtemp;*
emit(E.place ':=' E_1.place '+' E_2.place) }

E→ E_1 * E_2 { *E.place := newtemp;*
emit(E.place ':=' E_1.place '' E_2.place)* }

E→ - E_1 { *E.place := newtemp;*
emit(E.place ':=' 'unimus' E_1.place) }

E→ (E_1) { *E.place:=E_1.place)* }

E→ id { *p:=lookup(id.name);*
if p <> nil then E.place := p else error }

Hình 8.14 - *Lược đồ dịch sinh mã lệnh ba địa chỉ cho lệnh gán*

Hàm lookup tìm trong bảng ký hiệu xem có hay không một tên được cho bởi id.name. Nếu có thì trả về con trỏ của ô, nếu không trả về nil.

Xét luật sinh D → proc id ; ND1 ; S

Như trên đã nói, hành vi kết hợp với ký hiệu chưa kết thúc N cho phép con trỏ của bảng ký hiệu cho chương trình con đang nằm trên đỉnh Stack tblptr.

Các tên trong lệnh gán sinh ra bởi ký hiệu chưa kết thúc S sẽ được khai báo trong chương trình con này hoặc trong bao của nó. Khi tham khảo tới một tên thì trước hết hàm lookkup sẽ tìm xem có tên đó trong bảng ký hiệu hiện hành hay không. (Bảng danh biểu hiện hành được trỏ bởi top(tblptr)). Nếu không thì dùng con trỏ ở trong header của bảng để tìm bảng ký hiệu bao nó và tìm tên trong đó. Nếu tên không được tìm thấy trong tất cả các mức thì lookup trả về nil.

2. Địa chỉ hóa các phần tử của mảng

Các phần tử của mảng có thể truy xuất nhanh nếu chúng được liền trong một khối các ô nhớ kết tiếp nhau. Trong mảng một chiều nếu kích thước của một phần tử là w thì địa chỉ tương đối phần tử thứ i của mảng A được tính theo công thức

Địa chỉ tương đối của A[i] = base + (i-low) * w

Trong đó

low: là cận dưới tập chỉ số

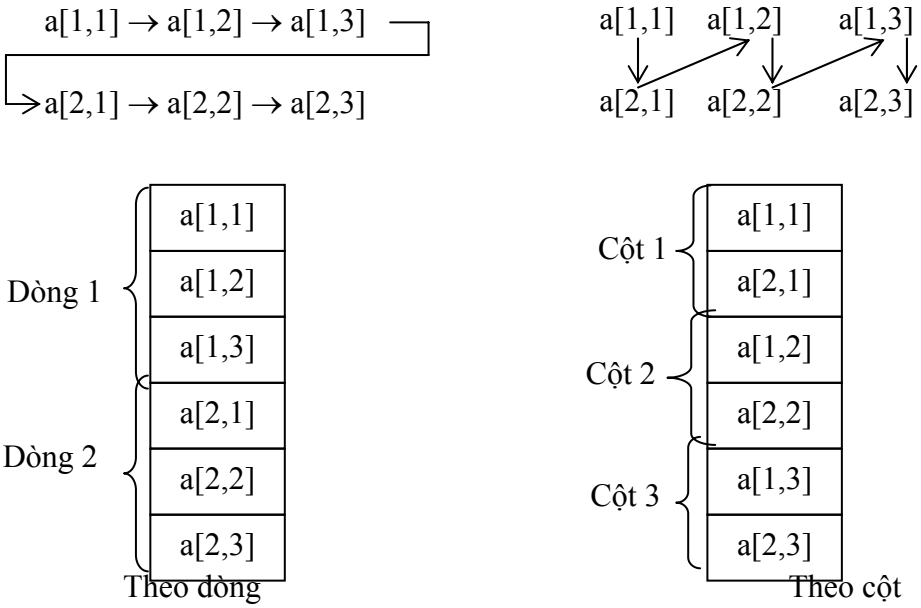
base: là địa chỉ tương đối của ô nhớ cấp phát cho mảng tức là địa chỉ tương đối của A[low]

Biến đổi một chút ta được

Địa chỉ tương đối của A[i]= i * w + (base -low * w)

Trong đó: c=base - low * w có thể tính được tại thời gian dịch và lưu trong bảng ký hiệu. Do đó địa chỉ tương đối A[i] = i * w +c.

Mảng hai chiều co ïthể xem như là một mảng theo một trong hai dạng: theo dòng (row_major) hoặc theo cột (colum_major)



Hình 8.15 - *Những cách sắp xếp của mảng hai chiều*

Trong trường hợp lưu trữ theo dòng, địa chỉ tương đối của phần tử a[i1, j2] có thể tính theo công thức

Địa chỉ tương đối của A[i1, j2] = base + ((i1- low1) * n_2 +j_2 -low_2) * w

Trong đó low1 và low2 là cận dưới của hai tập chỉ số.

n2 : là số các phần tử trong một dòng. Nếu gọi high2 là cận trên của tập chỉ số thứ 2 thì n2 = high2 -low2 +1

Trong đó công thức trên chỉ có i1, i2 là chưa biết tại thời gian dịch. Do đó, nếu biến đổi công thức để được :

*Địa chỉ tương đối của A[i1, j2]= ((i1 * n2)+j2) * w +(base-((low_1* n_2)+low_2) * w)*

Trong đó

C= (base- ((low1 * n2) + low2) * w) được tính tại thời gian dịch và ghi vào trong bảng ký hiệu.

Tổng quát hóa cho trường hợp k chiều, ta có

Địa chỉ tương đối của A[i1, i2, ... ik] là

((...((i_1n_2 + i_2) n_3 +i_3)...) n_k+i_k) w+base-((...((low_1n_2 + low_2) n_3+low_3)...)n_k+ low_k) w

3. Biến đổi kiểu trong lệnh gán

Giả sử chúng ta có 2 kiểu là integer và real; integer phải đổi thành real khi cần thiết. Ta có, các hành vi ngữ nghĩa kết hợp với luật sinh E → E_1 + E2 như sau:

E.place := newtemp

if E_1.type= integer **and** E_2.type = integer **then begin**

emit(E.place ':=' E_1.place 'int + ' E_2.place);

E.type:= integer;

end

else if E_1.type=real **and** E_2.type =real **then begin**

emit(E.place ':=' E_1.place 'real + ' E_2.place);

E.type:= real;

end

else if E_1.type=integer **and** E_2.type =real **then begin**

u:=newtemp; emit(u ':=' 'intoreal' E_1.place);

emit(E.place ':=' u 'real +' E_2.place);

E.type:= real;

end

else if E_1.type=real **and** E_2.type =integer **then begin**

u:=newtemp; emit(u ':=' 'intoreal' E_2.place);

emit(E.place ':= ' E_1.place 'real +' u);

E.type:= real;

end

else E.type := type_error;

end

Hình 8.16 - *Hành vi ngữ nghĩa của* $E \rightarrow E1 + E2$

Ví dụ 8.5: Với lệnh gán x := y + i * j trong đó x,y được khai báo là real; i , j được khai báo là integer. Mã lệnh 3 địa chỉ xuất ra là:

t1 := i int * j

t3 := intoreal t1

t2 := y real + t3

x := t2

IV. BIỂU THỨC LOGIC

Biểu thức logic được sinh ra bởi văn phạm sau:

$E \rightarrow E$ or $E \mid E$ and $E \mid$ not $E \mid (E) \mid$ id relop id $\mid$ true $\mid$ false

Trong đó or và and kết hợp trái; or có độ ưu tiên thấp nhất, kế tiếp là and và sau cùng là not

Thông thường có 2 phương pháp chính để biểu diễn giá trị logic.

Phương pháp 1: Mã hóa true và false bằng các số và việc đánh giá biểu thức được thực hiện tương tự như đối với biểu thức số học, có thể biểu diễn true bởi 1 , false bởi 0; hoặc các số khác không biểu diễn true, số không biểu diễn false...

Phương pháp 2: Biểu diễn giá trị của biểu thức logic bởi một vị trí đến trong chương trình. Phương pháp này rất thuận lợi để cài đặt biểu thức logic trong các điều khiển.

1. Biểu diễn số

Sử dụng 1 để biểu diễn true và 0 để biểu diễn false. Biểu thức được đánh giá từ trái sang phải theo cách tương tự biểu thức số học.

Ví dụ 8.6: Với biểu thức **a or b and not c**, ta có dãy lệnh 3 địa chỉ:

t1 := not c

t2 := b and t1

t3 := a or t2

Biểu thức quan hệ a<b tương đương lệnh điều kiện if a<b then 1 else 0. dãy lênh 3 địa chỉ tương ứng là

100 : if a<b goto 103

101 : t := 0

102 : goto 104

103 : t :=1

104 :

Ta có, lược đồ dịch để sinh ra mã lệnh 3 địa chỉ đối với biểu thức logic:

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

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

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

$E \rightarrow id_1$ relop id_2 { *E.place:= newtemp;*

emit('if' id_1.place relop.op id_2.place 'goto' nextstat +3);

emit(E.place ':=' '0'); emit('goto' nextstat +2);

emit(E.place ':=' '1') }

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

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

Hình 8.17 - *Lược đồ dịch sử dụng biểu diễn số để sinh mã lệnh ba địa chỉ cho các biểu thức logic*

Ví dụ 8.7: Với biểu thức **a < b or c< d and e < f**, nó sẽ sinh ra lệnh địa chỉ như sau:

100 : if a<b goto 103

101 : t1 := 0

102 : goto 104

103 : t1 := 1

104 : if c<d goto 107

105 : t2 := 0

106 : goto 108

107 : t2 := 1

```
108   : if e<f goto 111
109   : t3 := 0
110   : goto 112
111   : t3 := 1
112   : t4 := t2 and t3
113   : t5 := t1 or t4
```

Hình 8.18 - *Sự biên dịch sang mã lệnh ba địa chỉ cho a<b or c<d and e<f*

1. Mã nhảy

Đánh giá biểu thức logic mà không sinh ra mã lệnh cho các toán tử or, and và not. Chúng ta chỉ biểu diễn giá trị môt biểu thức bởi vị trí trong chuỗi mã. Ví dụ, trong chuỗi mã lệnh trên, giá trị t1 sẽ phụ thuộc vào việc chúng ta chọn lệnh 101 hay lệnh 103. Do đó giá trị của t1 là thừa.

2. Các lệnh điều khiển

S→ if E then S_1

| if E then S_1 else S_2

| while E do S_1

Với mỗi biểu thức logic E, chúng ta kết hợp với 2 nhãn

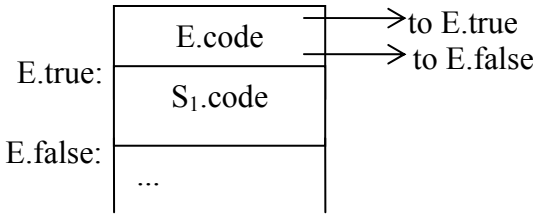
E.true : Nhãn của dòng điều khiển nếu E là true.

E.false : Nhãn của dòng điều khiển nếu E là false.

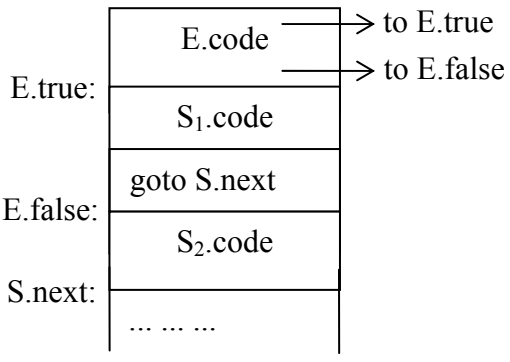
S.code : Mã lệnh 3 địa chỉ được sinh ra bởi S.

S.next : Là nhãn mà lệnh 3 địa chỉ đầu tiên sẽ thực hiện sau mã lệnh của S.

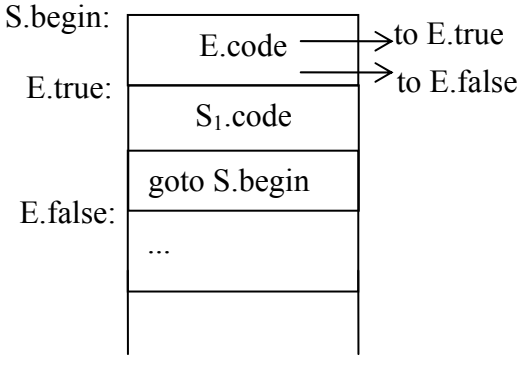
S.begin : Nhãn chỉ định lệnh đầu tiên được sinh ra cho S.



(a) if -then



(b) if -then-else



(c) while-do

Hình 8.19 - *Mã lệnh của các lệnh if-then, if-then-else, và while-do*

Ta có định nghĩa trực tiếp cú pháp cho các lệnh điều khiển

Luật sinh	Luật ngữ nghĩa
$S \rightarrow$ if E then S_1	*E.true := newlabel;* *E.false := S.next;* *S_1.next := S.next;* *S.code := E.code || gen(E.true ':') || S_1.code*
$S \rightarrow$ if E then S_1 else S_2	*E.true := newlabel;* *E.false := newlabel;* *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*
$S \rightarrow$ while E do S_1	*S.begin := newlabel;* *E.true := newlabel;* *E.fasle := S.next;* *S_1.next := S.begin;* *S.code:= gen(S.begin':') || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)*

Hình 8.20 - *Định nghĩa trực tiếp cú pháp của dòng điều khiển*

3. Dịch biểu thức logic trong các lệnh điều khiển

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

  if a<b goto E.true

  goto E.false

- Nếu E có dạng **E_1 or E_2**. Nếu E_1 là true thì E là true. Nếu E_1 là flase thì phải đánh giá E_2. Do đó E_1.false là nhãn của lệnh đầu tiên của E_2. E sẽ true hay false phụ thuộc vào E_2 là true hay false.
- Tương tự cho **E_1 and E_2**.
- Nếu E có dạng **not E_1** thì E_1 là true thì E là false và ngược lại.

Ta có định nghĩa trực tiếp cú pháp cho việc dịch các biểu thức logic thành mã lệnh 3 địa chỉ. Chú ý true và false là các thuộc tính kế thừa.

Luật sinh	Luật ngữ nghĩa
$E \rightarrow E_1$ or E_2	*E_1.true := E.true;* *E_1.false := newlabel;* *E_2.true := E.true;*

	E_2.false := E.false;
	E.code := E_1.code || gen(E.false ':') || E_2.code
E→ E_1 and E_2	E_1.true := newlabel;
	E_1.false := E.false;
	E_2.true := E.true;
	E_2.false := E.false;
	E.code := E_1.code || gen(E.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)

Hình 8.21 - *Định nghĩa trực tiếp cú pháp sinh mã lệnh ba địa chỉ cho biểu thức logic*

4. Biểu thức logic và biểu thức số học

Trong thực tế biểu thức logic thường chứa những biểu thức số học như *(a+b) < c.* Trong các ngôn ngữ mà false có giá trị số là 0 và true có giá trị số là 1 thì (a<b) + (b<a) có thể được xem như là một biểu thức số học có giá trị 0 nếu a = b và có giá trị 1 nếu a <> b.

Phương pháp biểu diễn biểu thức logic bằng mã lệnh nhảy có thể vẫn còn được sử dụng.

Xét văn phạm $E \rightarrow E + E \mid E$ *and* $E \mid E$ *relop* $E \mid id$

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

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

Ta có luật ngữ nghĩa kết hợp với E → E1 + E2 như sau

E.type := arith;

if E_1.type = arith **and** E_2.type = arith **then begin**

/ phép cộng số học bình thường */*

E.place := newtemp;

E.code := E_1.code \|\| E_2.code \|\| gen(E.place ':=' E_1.place '+' E_2.place)

end

else if E_1.type = arith **and** E_2.type = bool **then begin**

E.place := newtemp;

E_2.true := newlabel;

E_2.false := newlabel;

E.code := E_1.code || E_2.code || gen(E_2.true ':' E.place ':= ' E_1.place +1) ||

gen('goto' nextstat +1) || gen(E_2.false ':' E.place ':= ' E_1.place)

else if ...

Hình 8.22 - *Luật ngữ nghĩa cho luật sinh E → E1 +E2*

Trong trường hợp nếu có biểu thức logic nào có biểu thức số học, chúng ta sinh mã lệnh cho E1, E2 bởi các lệnh

E_2.true : E.place := E_1.place +1

goto nextstat +1

E_2.false : E.place := E_1.place

V. LỆNH CASE

Lệnh CASE hoặc SWITCH thường được sử dụng trong các ngôn ngữ lập trình.

1. Cú pháp của lệnh SWITCH/ CASE

SWITCH E

begin

case V_1 : S_1

case V_2 : S_2

....

case V_{n-1} : S_{n-1}

default: S_n

end

Hình 8.23 - *Cú pháp của câu lệnh switch*

2. Dịch trực tiếp cú pháp lệnh Case

1. Đánh giá biểu thức.
2. Tùy một giá trị trong danh sách các case bằng giá trị của biểu thức. Nếu không tìm thấy thì giá trị default của biểu thức được xác định.
3. Thực hiện các lệnh kết hợp với giá trị tìm được để cài đặt.

Ta có phương pháp cài đặt như sau

mã lệnh để đánh giá biểu thức E vào t

goto test

L1 : mã lệnh của S1

goto next

L2: mã lệnh của S2

goto next

....

Ln-1 : mã lệnh của Sn-1

goto next

Ln : mã lệnh của Sn

goto next

test : if t=V_1 goto L_1

if t=V_2 goto L_2

.. ..

if t=V_{n-1} goto L_{n-1}

else goto L_n

next:

Hình 8.24 - *Dịch câu lệnh case*

Một phương pháp khác để cài đặt lệnh SWITCH là

mã lệnh để đánh giá biểu thức E vào t

if t <> V1 goto L1

mã lệnh của S1

goto next

L_1 : if t <> V2 goto L_2

mã lệnh của S2

goto next

L_2:..... ..

L_{n-2} : if t <> V_{n-1} goto L_{n-1}

mã lệnh của Sn-1

goto next

Ln-1 : mã lệnh của Sn

next:

Hình 8.24 - *Một cách dịch khác của câu lệnh case*

BÀI TẬP CHƯƠNG VIII

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

a) Cây cú pháp.

b) Ký pháp hậu tố.

c) Mã lệnh máy 3 - địa chỉ.

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

a) Bộ tứ .

b) Bộ tam.

c) Bộ tam gián tiếp.

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

a) x = 1

b) x = y

c) x = x + 1

d) x = a + b * c

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

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

a) x = a [i] + 11

b) a [i] = b [c[j]]

c) a [i][j] = b [i][k] * c [k][j]

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

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

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

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