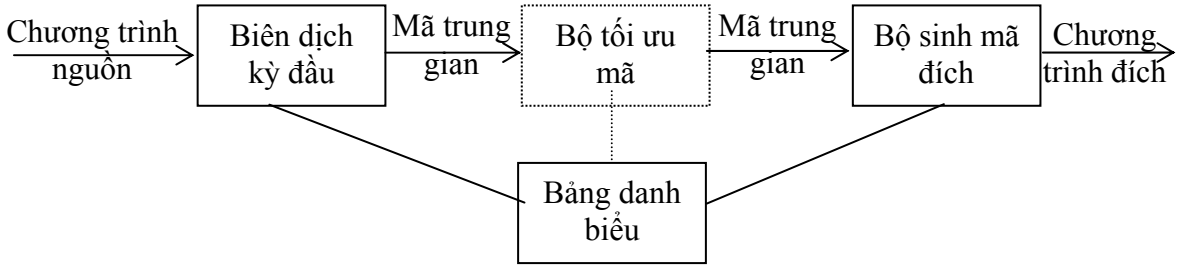
CHƯƠNG IX
SINH MÃ ĐÍCH

Nội dung chính:

Giai đoạn cuối của quá trình biên dịch là *sinh mã đích*. Dữ liệu nhập của bộ sinh mã đích là biểu diễn trung gian của chương trình nguồn và dữ liệu xuất của nó là một chương trình đích (hình 9.1). Kỹ thuật sinh mã đích được trình bày trong chương này không phụ thuộc vào việc dùng hay không dùng giai đoạn tối ưu mã trung gian .



Hình 9.1- *Vị trí của bộ sinh mã đích*

Nhìn chung một *bộ sinh mã đích* phải đảm bảo chạy hiệu quả và phải tạo ra chương trình đích đúng sử dụng hiệu quả tài nguyên của máy đích. Về mặt lý thuyết, vấn đề sinh mã tối ưu là không thực hiện được. Trong thực tế, ta có thể chọn những kỹ thuật heuristic để tạo ra mã tốt nhưng không nhất thiết là mã tối ưu. Chương này đề cập đến *các vấn đề cần quan tâm khi thiết kế một bộ sinh mã*. Bên cạnh đó một *bộ sinh mã đích đơn giản* từ chuỗi các lệnh ba địa chỉ cũng được giới thiệu.

Mục tiêu cần đạt:

Sau khi học xong chương này, sinh viên phải:

- Nắm được các vấn đề cần chú ý khi thiết kế bộ sinh mã đích.
- Biết cách tạo ra một bộ sinh mã đích đơn giản từ chuỗi các mã lệnh ba địa chỉ. Từ đó có thể mở rộng bộ sinh mã này cho phù hợp với ngôn ngữ lập trình cụ thể.

Kiến thức cơ bản:

Sinh viên phải có kiến thức về kiến trúc máy tính đặc biệt là phần hợp ngữ (assembly language) để thuận tiện cho việc tiếp nhận kiến thức về máy đích.

Tài liệu tham khảo:

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

[2] **Design of Compilers : Techniques of Programming Language Translation** - Karen A. Lemone - CRC Press, Inc, 1992.

I. CÁC VẤN ĐỀ THIẾT KẾ BỘ SINH MÃ

Trong khi thiết kế bộ sinh mã, các vấn đề chi tiết như quản trị bộ nhớ, chọn chỉ thị cấp phát thanh ghi và đánh giá thứ tự thực hiện ... phụ thuộc rất nhiều vào ngôn ngữ đích và hệ điều hành.

1. Dữ liệu vào của bộ sinh mã

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

2. Dữ liệu xuất của bộ sinh mã – Chương trình đích

Giống như mã trung gian, dữ liệu xuất của bộ sinh mã có thể ở một trong các dạng: Mã máy tuyệt đối, mã máy khả định vị địa chỉ hoặc hợp ngữ .

Việc tạo ra một chương trình đích ở dạng mã máy tuyệt đối cho phép chương trình này được lưu vào bộ nhớ và được thực hiện ngay.

Nếu chương trình đích ở dạng mã máy khả định vị địa chỉ (module đối tượng) thì hệ thống cho phép các chương trình con được biên dịch riêng rẽ. Một tập các module đối tượng có thể được liên kết và tải vào bộ nhớ để thực hiện nhờ bộ tải liên kết (linking loader). Mặc dù ta phải trả giá về thời gian cho việc liên kết và tải vào bộ nhớ các module đã liên kết nếu ta tạo ra các module đối tượng khả định vị địa chỉ. Nhưng bù lại, ta có sự mềm dẻo về việc biên dịch các chương trình con riêng rẽ và có thể gọi một chương trình con đã được biên dịch trước đó từ một module đối tượng. Nếu mã đích không tự động tái định vị địa chỉ, trình biên dịch phải cung cấp thông tin về tái định cho bộ tải (loader) để liên kết các chương trình đã được biên dịch lại với nhau.

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

3. Lựa chọn chỉ thị

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

MOV y, R0 / Lưu y vào thanh ghi Ro */*

ADD z, R0 / cộng z vào nội dung Ro, kết quả chứa trong R_o */*

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 này 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.

4. Cấp phát thanh ghi

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

1. Trong khi cấp phát thanh ghi, ta lựa chọn tập các biến lưu 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.

1. Lựa chọn cho việc đánh giá thứ tự

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

2. Sinh mã

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

II. MÁY ĐÍCH

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

Máy đích của chúng ta là máy tính địa chỉ byte với mỗi từ gồm bốn byte và có n thanh ghi : R0, R1 ... Rn-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 R0, M : Lưu nội dung của thanh ghi R0 vào vị trí nhớ M .
2. MOV 4(R0), M : Xác định một địa chỉ mới bằng cách lấy độ dời tương đối (offset) 4 cộng với nội dung của R0, sau đó lấy nội dung tại địa chỉ này, contains(4 + contains(R0)), lưu vào vị trí nhớ M.
3. MOV * 4(R0) , M : Lưu giá trị contents (contents (4 + contents (R0))) vào vị trí nhớ M.
4. MOV #1, R0 : Lấy hằng 1 lưu vào thanh ghi R0.

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 R0, R1 : Sao chép nội dung thanh ghi R0 vào thanh ghi R1. Chỉ thị này có giá là một vì nó chỉ chiếm một từ trong bộ nhớ .

2. MOV R5, M: Sao chép nội dung thanh ghi R5 vào vị trí nhớ M. Chỉ thị này có giá trị là hai vì địa chỉ của vị trí nhớ M là một từ sau chỉ thị.

3. Chỉ thị ADD #1, R3: cộng hằng 1 vào nội dung thanh ghi 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(R0), *12 (R1) : Lưu giá trị của contents (contents (12 + contents (R_1))) - contents (4 + contents (R0)) vào đích *12(R1). Giá của chỉ thị này là ba vì hằng 4 và 12 được lưu trữ trong hai từ kế tiếp theo sau chỉ thị.

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

1. *MOV b, R_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.

III. QUẢN LÝ BỘ NHỚ TRONG THỜI GIAN THỰC HIỆN

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 bày 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 này, ta minh họa các chiến lược cấp phát sử dụng trường trạng thái để giữ giá trị trả về và dữ liệu cục bộ, các trường còn lại được dùng như đã đề cập ở chương VII.

Việc cấp phát và giải phóng các 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:

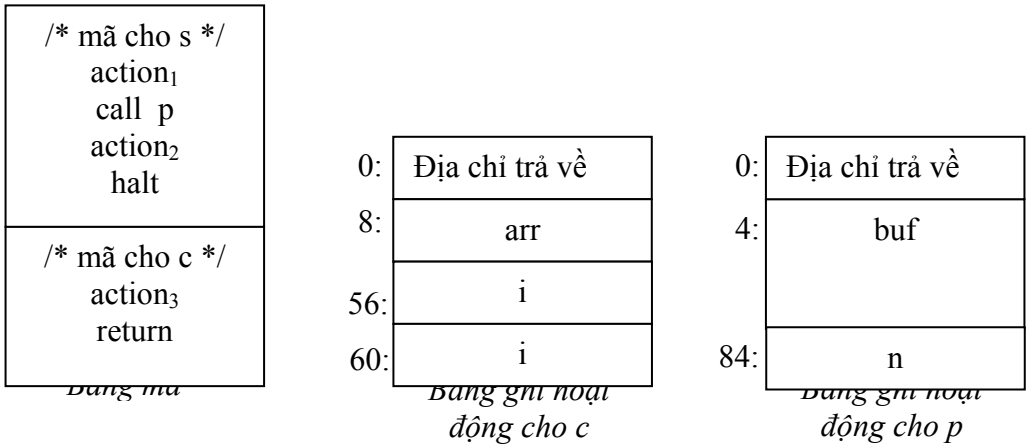
1. call

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 – *Dữ liệu 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 dữ liệu vào của hình 9.2*

Sự thực hiện bắt đầu bằng chỉ thị action tại địa chỉ 100. Chỉ thị MOV ở địa chỉ 120 sẽ lưu địa chỉ trả về 140 vào trường trạng thái máy, là từ đầu tiên trong 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 này 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 */**

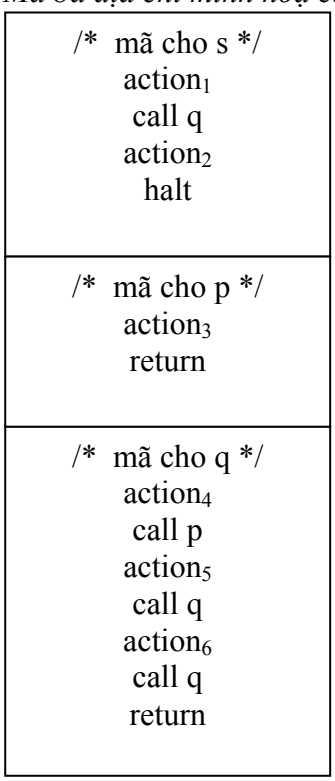
SUB #caller.recordsize, SP

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 này 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:

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 action$_4$ gồm lệnh nhảy có điều kiện tới địa chỉ 456 có lệnh trả về từ q. Ngược lại chương trình đệ quy q có thể gọi chính nó mãi. Trong ví dụ này chúng ta giả sử lần gọi đầu tiên trên q sẽ không trả về chương trình gọi ngay, nhưng những lần sau thì có thể. SP có giá trị lúc đầu là 600, địa chỉ bắt đầu của Stack. SP lưu giữ giá trị 620 chỉ trước khi chuyển quyền điều khiển từ s sang q vì kích thước của 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   #0, 12(R3)

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

IV. KHỐI CƠ BẢN VÀ LƯU ĐỒ

Đồ thị biểu diễn các lệnh ba địa chỉ, được gọi là lưu đồ, giúp ta hiểu các giải thuật sinh mã ngay cả khi đồ thị không được xác định cụ thể bằng giải thuật sinh mã. Các nút của lưu đồ biểu diễn sự tính toán, các cạnh biểu diễn dòng điều khiển.

1. Khối cơ bản

Khối cơ bản (basic block) là chuỗi các lệnh kế tiếp nhau trong đó dòng điều khiển đi vào lệnh đầu tiên của khối và ra ở lệnh cuối cùng của khối mà không bị dừng hoặc rẽ nhánh. Ví dụ chuỗi lệnh ba địa chỉ sau tạo nên một khối cơ bản

$t_1 := a * a$

$t_2 := a * b$

$t_3 := 2 * t_2$

$t_4 := t_1 + t_2$

$t_5 := b * b$

$t_6 := t_4 + t_5$

Lệnh ba địa chỉ x := y + z dùng các giá trị được chứa ở các vị trí nhớ của y, z để thực hiện phép cộng và xác định địa chỉ của x để lưu kết quả phép cộng vào. Một tên trong khối cơ bản được gọi là 'sống' tại một điểm nào đó nếu giá trị của nó sẽ được sử dụng sau điểm đó trong chương trình hoặc được dùng ở khối cơ bản khác. Giải thuật sau đây phân chia chuỗi các lệnh ba địa chỉ sang các khối cơ bản.

- **Giải thuật 9.1: Phân chia các khối cơ bản**

Input: Các lệnh ba địa chỉ.

Output: Danh sách các khối cơ bản với từng chuỗi các lệnh ba địa chỉ cho từng khối.

Phương pháp:

1. Xác định tập các lệnh dẫn đầu (leader), các lệnh đầu tiên của các khối cơ bản, ta dùng các quy tắc sau:

i) Lệnh đầu tiên là lệnh dẫn đầu.

ii) Bất kỳ lệnh nào là đích nhảy đến của lệnh GOTO có điều kiện hoặc không điều kiện đều là lệnh dẫn đầu.

iii) Bất kỳ lệnh nào đi sau lệnh GOTO có điều kiện hoặc không điều kiện đều là lệnh dẫn đầu.

2. Với mỗi lệnh dẫn đầu, khối cơ bản gồm có nó và tất cả các lệnh tiếp theo nhưng không gồm một lệnh dẫn đầu nào khác hay là lệnh kết thúc chương trình.

Ví dụ 9.3: Đoạn chương trình sau tính tích vectơ vô hướng của hai vectơ a và b có độ dài 20.

```
Begin
        prod := 0
        i := 1
        Repeat
                prod: = prod + a [i] * b[i];
```

```
            i := i + 1
        Until  i > 20
End
```

Hình 9.6 - *Chương trình tính tích vectơ vô hướng*

Đoạn chương trình này được dịch sang mã ba địa chỉ như sau:

(1) $prod := 0$

(2) $i := 1$

(3) $t_1 := 4 * i$

(4) $t2 := a[t1]$ /* tính a[i] */

(5) $t_3 := 4 * i$

(6) $t_4 := b[t_3]$

(7) $t_5 := t_2 * t_4$

(8) $t_6 := prod + t_5$

(9) $prod := t_6$

(10) $t_7 := i + 1$

(11) $i := t_7$

(12) $if \ \ i<=20 \ \ goto \ (3)$

Hình 9.7 - *Mã ba địa chỉ để tính tích vectơ vô hướng*

Lệnh (1) là lệnh dẫn đầu theo quy tắc i, lệnh (3) là lệnh dẫn đầu theo quy tắc ii và lệnh sau lệnh (12) là lệnh dẫn đầu theo quy tắc iii.

Như vậy lệnh (1) và (2) tạo nên khối cơ bản thứ nhất . Lệnh (3) đến (12) tạo nên khối cơ bản thứ hai.

2. Sự chuyển đổi giữa các khối

Khối cơ bản tính các biểu thức. Các biểu thức này là giá trị của các tên "sống" khi ra khỏi khối. Hai khối cơ bản tương đương nhau khi chúng tính các biểu thức giống nhau.

Một số chuyển đổi có thể được áp dụng vào một khối cơ bản mà không làm thay đổi các biểu thức được tính toán trong đó. Nhiều phép chuyển đổi rất có ích vì nó cải thiện chất lượng mã đích được sinh ra từ khối cơ bản. Hai phương pháp chuyển đổi cục bộ quan trọng được áp dụng cho các khối cơ bản là chuyển đổi bảo toàn cấu trúc và chuyển đổi đại số.

Chuyển đổi bảo toàn cấu trúc

Những chuyển đổi bảo toàn cấu trúc trên các khối cơ bản bao gồm:

1. Loại bỏ các biểu thức con chung.
2. Loại bỏ mã chết .
3. Đặt tên lại các biến tạm.
4. Hoán đổi hai lệnh độc lập kề nhau.

Giả sử trong các khối cơ bản không chứa dãy, con trỏ hay lời gọi chương trình con.

1. *Loại bỏ các biểu thức con chung*

Khối cơ bản sau:

a := b + c

b := a - d

c := b + c

d := a - d

Câu lệnh thứ hai và thứ tư tính cùng một biểu thức b + c - *d. Vì* vậy, khối cơ bản này được chuyển thành khối tương đương sau:

a := b + c

b := a - d

c := b + c

d := b

2. *Loại bỏ mã lệnh chết*

Giả sử x không còn được sử dụng nữa. Nếu câu lệnh x := y + z xuất hiện trong khối cơ bản thì lệnh này sẽ bị loại mà không làm thay đổi giá trị của khối.

3. *Đặt lại tên cho biến tạm*

Giả sử ta có lệnh t := b + c với t là biến tạm. Nếu ta viết lại lệnh này thành u := b + c mà u là biến tạm mới và thay t bằng u ở bất cứ chỗ nào xuất hiện t thì giá trị của khối cơ bản sẽ không bị thay đổi. Thực tế, ta có thể chuyển một khối cơ bản sang một khối cơ bản tương đương. Và ta gọi khối cơ bản được tạo ra như vậy là dạng chuẩn.

Giả sử chúng ta một khối với hai câu lệnh kế tiếp:

$t_1 := b + c$

$t_2 := x + y$

Ta có thể hoán đổi hai lệnh này mà không làm thay đổi giá trị của khối nếu và chỉ nếu x và y đều không phải t1 cũng như b và c đều không phải là t2. Khối cơ bản có dạng chuẩn cho phép tất cả các lệnh có quyền hoán đổi nếu có thể.

Chuyển đổi đại số

Các biểu thức trong một khối cơ bản có thể được chuyển đổi sang các biểu thức tương đương. Phép chuyển đổi đại số này giúp ta đơn giản hoá các biểu thức hoặc thay thế các biểu thức có giá cao bằng các biểu thức có giá rẻ hơn.

Chẳng hạn, câu lệnh *x := x + 0* hoặc *x := x * 1* có thể được loại bỏ khỏi khối mà không làm thay đổi giá trị của biểu thức. Toán tử lũy thừa trong câu lệnh x := y ** 2 cần một lời gọi hàm để thực hiện. Tuy nhiên, lệnh này vẫn có thể được thay bằng lệnh tương đương có giá rẻ hơn mà không cần lời gọi hàm.

3. Lưu đồ

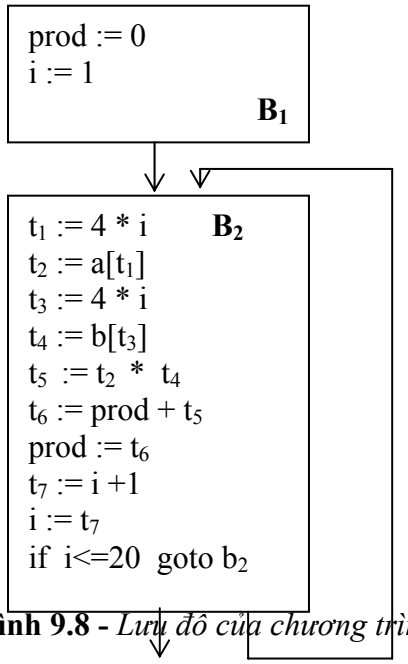
Ta có thể thêm thông tin về dòng điều khiển vào tập các khối cơ bản bằng việc xây dựng các đồ thị trực tiếp được gọi là lưu đồ (flew graph). Các nút của lưu đồ là khối cơ bản. Một nút được gọi là khởi đầu nếu nó chứa lệnh đầu tiên của chương trình. Cạnh nối trực tiếp từ khối B_1 đến khối B_2 nếu B_2 là khối đứng ngay sau B_1 trong một chuỗi thực hiện nào đó. Nghĩa là, nếu:

1. Lệnh nhảy không hoặc có điều kiện từ lệnh cuối của B_1 sẽ đi đến lệnh đầu tiên của B_2.

2. B_2 đứng ngay sau B_1 trong thứ tự của chương trình và B_1 không kết thúc bằng một lệnh nhảy không điều kiện.

Chúng ta nói B_1 là tiền bối (predecessor) của B_2 hay B_2 là hậu duệ (sucecssor) của B_1.

Ví dụ 9.4:



Hình 9.8 - *Lưu đồ của chương trình*

4. Biểu diễn các khối cơ bản

Các khối cơ bản được biểu diễn bởi nhiều loại cấu trúc dữ liệu.

Sau khi phân chia các lệnh ba địa chỉ bằng giải thuật 9.1. Mỗi khối cơ bản được biểu diễn bằng một mẩu tin gồm một số bộ tứ , theo sau là một con trỏ trỏ tới lệnh dẫn đầu (bộ tứ đầu tiên) của khối, và một danh sách các tiền bối và hậu duệ của khối.

DAG cũng là một cấu trúc dữ liệu rất thích hợp để thực hiện việc chuyển đổi các khối cơ bản. Xây dựng DAG từ các lệnh ba địa chỉ là một cách tốt để xác định được: Các biểu thức chung (được tính nhiều lần), tên được dùng trong khối nhưng không được dùng khi ra ngoài khối, và các biểu thức mà giá trị của nó được dùng khi ra khỏi khối.

- **Giải thuật 9.2: Xây dựng DAG**

  Input: Khối cơ bản

  Output: DAG cho khối cơ bản, chứa các thông tin sau:

  1. Tên cho từng nút. Tên nút lá là danh biểu (hằng số). Tên nút trung gian là toán tử.
  2. Với mỗi nút, một danh sách (có thể rỗng) gồm các danh biểu (hằng số không được phép có trong danh sách này).

  Phương pháp: Giả sử ta đã có cấu trúc dữ liệu để tạo nút có một hoặc hai con. Ta phải phân biệt con bên trái và bên phải đối với những nút có hai con. Ta cũng có vị trí để ghi tên cho mỗi nút và có cơ chế tạo danh sách liên kết của các danh biểu gắn với mỗi nút. Ta cũng giả sử tồn tại hàm node (identifier), khi ta xây dựng DAG, sẽ trả về nút mới nhất có liên quan với identifier. Thực tế node (identifier) là nút biểu diễn giá trị của danh biểu (identifier) tại thời điểm hiện tại trong quá trình xây dựng DAG.

  Quá trình xây dựng DAG thực hiện qua các bước từ (1) đến (3) cho mỗi lệnh của khối. Lúc đầu, ta giả sử chưa có các nút và hàm node không được định nghĩa cho tất cả các đối số. Các dạng lệnh ba địa chỉ ở một trong các dạng sau: *(i) x := y op z, (ii) x := op y, (iii) x := y*.

Trường hợp lệnh điều kiện, chẳng hạn if i <= 20 goto, ta coi là trường hợp (i) với x không được định nghĩa.

1. Nếu node (y) không được định nghĩa, tạo lá có tên y và *node (y)* chính là nút này. Trong trường hợp (i), nếu node(z) không được định nghĩa, ta tạo lá tên z và lá chính là node (z).

2. Trong trường hợp (i) , xác định xem trên DAG có nút nào có tên op mà con trái là node (y) và con phải là node (z). Nếu không thì tạo ra nút có tên op, ngược lại n là nút đã tìm thấy hoặc đã được tạo. Trong trường hợp (ii) , ta xác định xem có nút nào có tên op, mà nó chỉ có một con duy nhất là node (y). Nếu chưa có nút như trên ta sẽ tạo nút op và coi n là nút tìm thấy hoặc vừa được tạo ra. Trong trường hợp thứ (iii) thì đặt n là *node(y)*.

3. Xoá x ra khỏi danh sách các danh biểu gán với nút node(x). Nối x vào danh sách các danh biểu gắn vào nút n được tìm ở bước (2) và đặt *node(x)* cho n.

5. Vòng lặp

Vòng lặp (loop) là một tập hợp các nút trong lưu đồ sao cho:

1. Tất cả các nút trong tập hợp phải kết nối chặt chẽ với nhau, nghĩa là phải có con đường với kích thước bằng một hoặc lớn hơn đi từ một nút bất kỳ trong vòng lặp đến một nút bất kỳ khác và nó phải nằm hoàn toàn trong vòng lặp.

2. Các nút lựa chọn này chỉ có duy nhất một lối vào, nghĩa là con đường từ nút ngoài vòng lặp đi vào vòng lặp phải đi qua lối vào đó.

Nếu một vòng lặp không chứa vòng lặp nào khác thì gọi là vòng lặp trong cùng.

V. THÔNG TIN SỬ DỤNG TIẾP

1. Tính toán sử dụng tiếp

Việc sử dụng tên trong lệnh ba địa chỉ được định nghĩa như sau:

Giả sử lệnh ba địa chỉ i gán một giá trị cho x. Nếu x là một toán hạng trong lệnh j, dòng điều khiển đi từ lệnh i đến j đồng thời dọc đường đi này ta không xen vào các lệnh gán cho x, thì ta nói rằng lệnh j dùng giá trị của x được tính toán tại i.

Trong chương này, ta chỉ đề cập đến việc tiếp tục sử dụng tên của một lệnh ba địa chỉ ngay trong khối chứa lệnh đó.

Giải thuật xác định những sử dụng tiếp theo tạo nên sự duyệt lùi đi qua các khối cơ bản. Ta có thể dễ dàng xác định lệnh ba địa chỉ cuối cùng của mỗi khối cơ bản. Vì các chương trình con có những hiệu ứng lề tùy ý nên ta giả sử rằng mỗi lời gọi chương trình con bắt đầu tại mỗi khối cơ bản mới.

Để tìm lệnh cuối cùng của một khối cơ bản, ta duyệt lùi tới phần đầu, lưu giữ các thông tin như: x có được tiếp tục sử dụng trong khối hay không? x có còn "sống" khi ra khỏi khối?

Giả sử rằng ta tới được lệnh ba địa chỉ i: ***x := y op z*** trong khi duyệt lùi. Ta có thể làm theo các như sau:

1. Gán cho lệnh i các thông tin được tìm thấy trong bảng danh biểu. Đó là những thông tin xác định x, y, z có được tiếp tục sử dụng? và còn "sống"?

2. Trong bảng danh biểu, đặt x không "sống" và không được dùng tiếp.

3. Trong bảng danh biểu, đặt y, z "sống" và được sử dụng tiếp. Chú ý rằng các bước 2 và 3 không thể xen kẽ nhau vì x có thể là y hoặc z.

Đối với lệnh ba địa chỉ dạng ***x := y*** hoặc ***x := op y,*** các bước làm tương tự như trên nhưng bỏ qua z.

2. Bộ nhớ của các tên tạm

Nhìn chung, ta có thể gói các biến trung gian vào cùng một vị trí nhớ nếu chúng không cùng "sống".

Hầu như tất cả các tên tạm (biến trung gian) được định nghĩa và được sử dụng trong các khối cơ bản, thông tin sử dụng tiếp có thể gói các biến trung gian. Trong chương này, ta không nói tới các biến trung gian được sử dụng trong nhiều khối.

Ta có thể cấp phát các vị trí nhớ cho các biến trung gian bằng cách kiểm tra kết quả trả về của mỗi biến và gán một biến trung gian vào vị trí đầu tiên trong trường của các biến trung gian, trường không chứa biến "sống". Nếu một biến trung gian không được gán cho bất cứ vị trí nào được tạo ra trước đó, thêm vị trí mới vào vùng dữ liệu của chương trình con hiện hành. Trong nhiều trường hợp, các biến trung gian được gói vào trong các thanh ghi hơn là vào các vị trí nhớ.

Chẳng hạn, sáu biến trung gian trong khối cơ bản sau được gói vào hai vị trí nhớ là t_1 và t_2:

$t_1 := a * a$

$t_2 := a * b$

$t_2 := 2 * t_2$

$t_1 := t_1 + t_2$

$t_2 := b * b$

$t_1 := t_1 + t_2$

VI. BỘ SINH MÃ ĐƠN GIẢN

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

(a) Thanh ghi đó được sử dụng cho sự tính toán khác

(b) Trước khi có lệnh gọi chương trình con, lệnh nhảy hoặc lệnh có nhãn.

Điều kiện (b) chỉ ra rằng bất cứ giá trị nào cũng phải được lưu vào bộ nhớ trước khi kết thúc một khối cơ bản. Vì sau khi ra khỏi khối cơ bản, ta có thể đi tới các khối khác hoặc ta có thể đi tới một khối xác định từ một khối khác. Trong trường hợp (a), ta không thể làm được điều này mà không giả sử rằng số lượng được dùng bởi khối xuất hiện trong cùng thanh ghi không có cách nào để đạt tới khối đó. Để tránh lỗi có thể xảy ra, giải thuật sinh mã đơn giản sẽ lưu giữ tất cả các giá trị khi đi qua ranh giới của khối cơ bản cũng như khi gọi chương trình con.

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

Nếu b ở trong 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 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 này, giải thuật sinh mã vừa được trình bày sẽ tạo chuỗi mã đích với giả sử rằng: a, b, c luôn ở trong bộ nhớ và t, u, v là các biến tạm không có trong bộ nhớ .

Câu lệnh 3 địa chỉ	Mã đích	Giá	Bộ mô tả thanh ghi	Bộ mô tả địa chỉ
t := a - b	MOV a, R_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 này cho đến khi lệnh ba địa chỉ cuối cùng d := v + u được xử lý. Chú ý rằng R_0 là rỗng vì u không còn được dùng nữa. Sau đó ta tạo ra chỉ thị, cuối cùng của khối, MOV R_0, d để lưu biến "sống" d. Giá của chuỗi mã đích được sinh ra như ở trên là 12. Tuy nhiên, ta có thể giảm giá xuống còn 11 bằng cách thay chỉ thị MOV a, R_1 bằng MOV R_0, R_1 và xếp chỉ thị này sau chỉ thị thứ nhất.

4. Sinh mã cho loại lệnh khác

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

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

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 Mi hoặc trên Stack tại vị trí Si 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 M_i**		**p trong Stack**	
	Mã	**Giá**	**Mã**	**Giá**	**Mã**	**Giá**
a:= *p	MOV *R_p, a	2	MOV M_p, R	3	MOV S_p(A), R	3
			MOV *R, R		MOV *R, R	
*p:= a	MOV a, *R_p	2	MOV M_p, R	4	MOV a, R	4
			MOV a, *R		MOV R, *S_p(A)	

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

Ba chuỗi mã đích tuỳ thuộc vào p ở trong thanh ghi Rp, hoặc p trong vị trí nhớ Mp, hoặc p ở trong Stack tại offset là Sp 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