Xây dựng
CHƯƠNG TRÌNH DỊCH

Phạm Đăng Hải
haipd@soict.hut.edu.vn

Chương 5: Sinh mã

1. Sinh mã trung gian
2. Sinh mã đích
3. Tối ưu mã

Giới thiệu

- Bộ sinh mã trung gian chuyển chương trình nguồn sang chương trình tương đương trong ngôn ngữ trung gian
  - Chương trình trung gian là một chương trình cho một máy trừu tượng
- Ngôn ngữ trung gian được người thiết kế trình biên dịch quyết định, có thể là:
  - Cây cú pháp
  - Ký pháp Ba Lan sau (hậu tố)
  - Mã 3 địa chỉ …

Nội dung

- Mã 3 địa chỉ
  - Các dạng mã,
  - Dịch trực tiếp cú pháp thành mã 3 địa chỉ
  - Cài đặt mã
- Sinh mã cho khai báo
  - Quy tắc ngữ nghĩa
  - Lưu trữ thông tin về phạm vi
- Sinh mã cho lệnh gán
  - Tên trong bảng ký hiệu
  - Địa chỉ hóa các phần tử của mảng
- Sinh mã cho các biểu thức logic
- Sinh mã cho các cấu trúc lập trình

Mã 3 địa chỉ

- Mã trung gian thường dùng : **mã ba địa chỉ**, tương tự mã assembly
- Chương trình trung gian là một dãy các lệnh thuộc kiểu mã 3 địa chỉ
  - Mỗi lệnh gồm tối đa 3 toán hạng
  - Tồn tại nhiều nhất một toán tử ở vế phải cộng thêm một toán tử gán
- x,y,z là các địa chỉ , tức là tên, hằng hay các tên trung gian do trình biên dịch sinh ra
  - Tên trung gian phải được sinh để thực hiện các phép toán trung gian
  - Các địa chỉ được thực hiện như con trỏ tới phần tử tương ứng của nó trong bảng ký hiệu

Mã 3 địa chỉ → Ví dụ

- Câu lệnh
  - A = x + y * z
- Chuyển thành mã 3 địa chỉ

  T = y * z

  A = x + T

T là tên trung gian

  - Được bộ sinh mã trung gian sinh ra cho các toán tử trung gian

Mã 3 địa chỉ → Các dạng phổ biến

- Mã 3 địa chỉ tương tự mã Assembly:
  - Lệnh có thể có nhãn,
  - Tồn tại những lệnh chuyển điều khiển cho các cấu trúc lập trình.
- Các dạng lệnh
  - *Lệnh gán* **x := y *op* z**.
  - *Lệnh gán với phép toán 1 ngôi* : **x := *op* y**.
  - *Lệnh sao chép*: **x := y.**
  - *Lệnh gán có chỉ số* ***X := y[i] hoặc x[i]= y***

Mã 3 địa chỉ → Các dạng phổ biến

- *Lệnh gán địa chỉ và con trỏ*

  x = &y; x = * y; *x = y

- *Lệnh nhảy không điều kiện*: **goto L**,
  - L là nhãn của một lệnh
- Lệnh nhảy có điều kiện **IF x *relop* y goto L**.
  - *Nếu thỏa mãn quan hệ relop (>,>=,<,..) thì thực hiện lệnh tại nhãn L,*
  - *Nếu không thỏa mãn, thực hiện câu lệnh ngay tiếp theo lệnh **IF***

Mã 3 địa chỉ → Các dạng phổ biến

- *Gọi thủ tục với n tham số* ***call p, n.***

  Khai báo tham số ***param x***

  Trả về giá trị ***return y***

Thường dung với chuỗi lệnh 3 địa chỉ

– *Lời gọi chương trình con* ***Call p(X_1, X_2,…x_n)*** *sinh ra*

param x_1

param x_2

param x_n

Call p, n

Chương trình dịch định hướng cú pháp

- Mỗi ký hiệu VP liên kết với một tập thuộc tính
  - Hai loại thuộc tính: Tổng hợp và kế thừa
- **Thuộc tính tổng hợp**
  - Giá trị của thuộc tính tại một nút trong cây được xác định từ giá trị của các nút con của nó.
- **Thuộc tính kế thừa**
  - Giá trị của thuộc tính được định nghĩa dựa vào giá trị nút cha và/hoặc các nút anh em của nó.

Dạng của định nghĩa hướng cú pháp

- Mỗi sản xuất dạng $A \rightarrow \alpha$ liên hệ với một tập luật ngữ nghĩa có dạng $b = f(c_1, c_2, \ldots\ldots, c_n)$ trong đó f là một hàm và b thoả một trong hai yêu cầu sau:
  - b là một thuộc tính tổng hợp của A và c_1,.., c_n là các thuộc tính liên kết với các ký hiệu trong về phải sản xuất $A \rightarrow \alpha$
  - b là một thuộc tính kế thừa một trong những ký hiệu xuất hiện trong α, và c_1,.., c_n là thuộc tính của các ký hiệu trong về phải sản xuất $A \rightarrow \alpha$
- Tập luật ngữ nghĩa dùng để tính giá trị thuộc tính của ký hiệu A

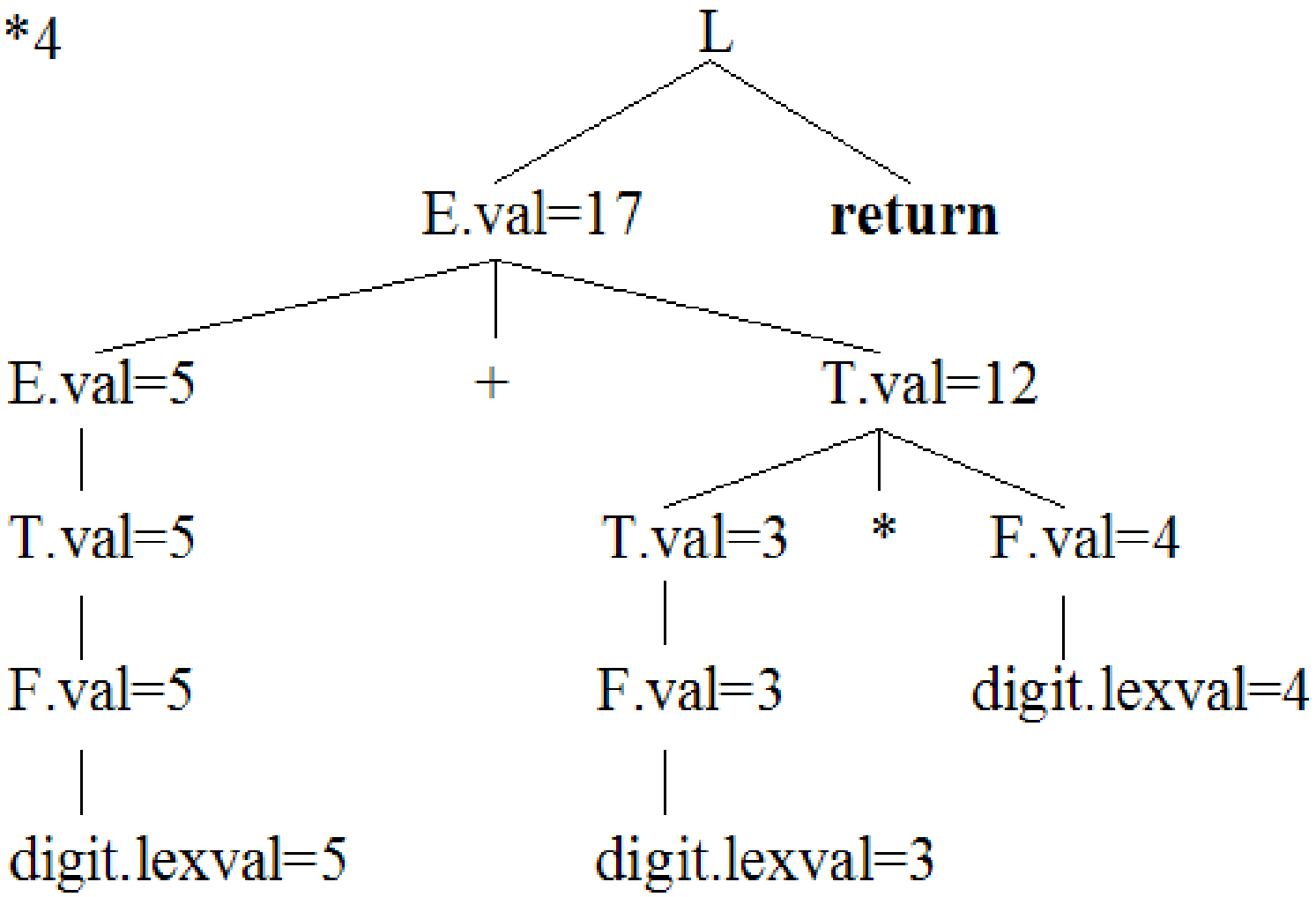
Ví dụ

Sản xuất	Quy tắc ngữ nghĩa
L → E **return**	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*

- *Các ký hiệu **E, T, F** có thuộc tính tổng hợp **val***
- *Từ tố **digit** có thuộc tính tổng hợp **lexval** (Được bộ phân tích từ vựng đưa ra)*

Chương trình dịch định hướng cú pháp

Input: 5+3*4



Dịch trực tiếp cú pháp thành mã 3 địa chỉ

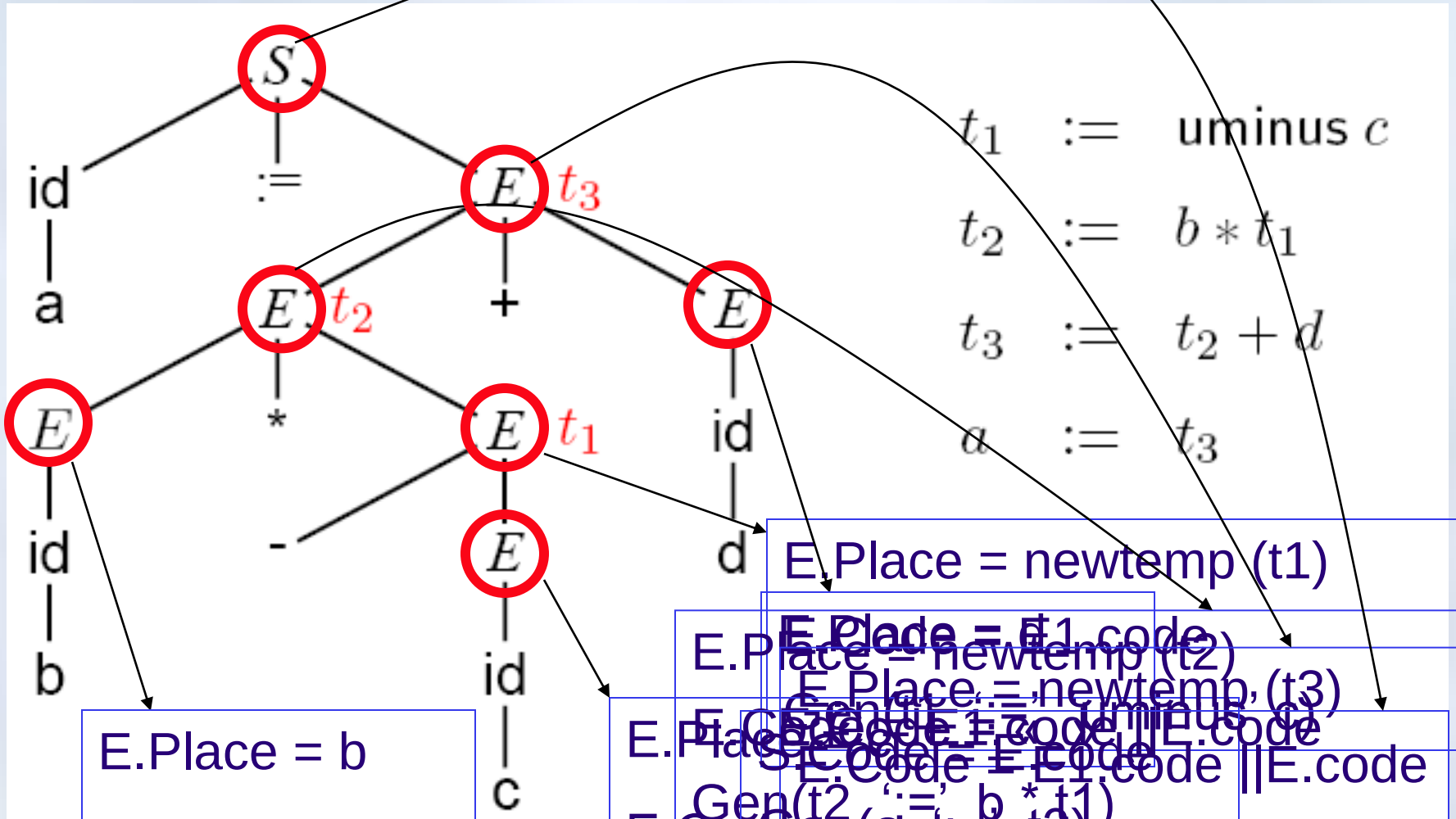
- Thuộc tính tổng hợp **S.code** biểu diễn mã ba địa chỉ của lệnh **S**
- Các tên trung gian được sinh ra cho các tính toán trung gian
- Các ký hiệu không kết thúc E có 2 thuộc tính
  - E.place Tên chứa giá trị của E
  - E.code là chuỗi mã lệnh địa chỉ để đánh giá E
- Hàm **newtemp()** sinh ra các tên trung gian t1, t2,. .
- Sử dụng hàm **gen(x ':=' y '+' z)** thể hiện mã 3 địa chỉ câu lệnh x := y + z
  - Các biểu thức ở các vị trí của x, y, z được đánh giá khi truyền vào hàm gen()

Dịch trực tiếp cú pháp thành mã 3 địa chỉ

Sản xuất	Quy tắc 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)	E.*place*= E_1.*place* ; E.*code* = E_1.*code*
E → **Id**	E.*place* = id.*place* ; E.*code* = ''

Dịch trực tiếp cú pháp thành mã 3 địa chỉ

Câu lệnh gán: a := b * -c + d



E.Place = b

E.Code = « »

E.Place = newtemp (t1)

E.Place = t1 code

E.Place = newtemp (t2)

E.Place = newtemp (t3)

E.Code = E1.code ||E.code

E.Code = E1.code ||E.code

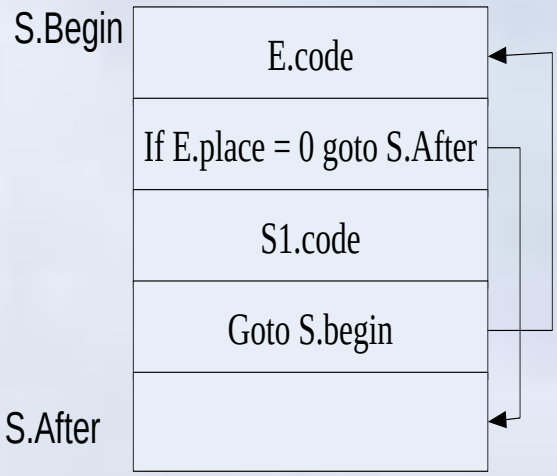
Gen(t2 ':=' b * t1)

Gen(a ':=' t3)

Gen(t3 ':=' t2 + d)

Dịch trực tiếp cú pháp thành mã 3 địa chỉ

Ví dụ: Câu lệnh lặp while

Sản xuất	**Quy tắc ngữ nghĩa**
S → while E do S1 	S.Begin = newLabel S.After = newLabel S.Code = /*Sinh mã cho lệnh while gồm*/ gen(S.begin ':') || E.code|| /*Sinh mã cho lệnh đánh giá E*/ Gen('if' E.place '=' 0 'goto' S.After)|| S1.code|| /*Sinh mã cho lệnh S1*/ gen('goto' S.Begin)|| /*Sinh mã cho goto*/ Gen(S.After ':') /*Sinh mã cho nhãn mới*/

Hàm newLabel: Sinh ra một nhãn mới

Cài đặt lệnh 3 địa chỉ→Biểu diễn bộ bốn

- Sử dụng cấu trúc gồm 4 trường: Op, Arg1, Arg2, Result
  - **Op:** Chứa mã nội bộ của toán tử
  - Các trường Arg1, Arg2, Result trỏ tới các ô trong bảng ký hiệu ứng với các tên tương ứng
- Câu lệnh dạng **a:= b Op c**
  - Đặt b vào Arg1, C vào Arg2 và a vào Result
- Câu lệnh một ngôi: **a:= b**; **a:=-b**
  - Không sử dụng Arg2

Cài đặt lệnh 3 địa chỉ→Biểu diễn bộ bốn

Ví dụ lệnh a = -b * (c+d)

Lệnh 3 địa chỉ

t1 := -b

t2 := c + d;

t3 := t1 * t2;

a := t3

Biểu diễn bởi dãy các bộ 4

	Op	Arg1	Arg2	Result
0	uminus	b		t1
1	+	c	d	t2
2	*	t1	t2	t3
3	:=	t3		a

Các tên tạm phải được đưa vào bảng ký hiệu

Cài đặt lệnh 3 địa chỉ→Biểu diễn bộ ba

- Mục đích để trách đưa tên tạm vào bảng ký hiệu
- Tham khảo tới giá trị tạm thời bằng vị trí lệnh sử dụng tính ra giá trị này
- Bỏ trường **Result**, Các trường **Arg1, Arg2** trỏ tới phần tử tương ứng trong bảng ký hiệu hoặc câu lệnh tương ứng

	Op	Arg1	Arg2
0	uminus	b	
1	+	c	d
2	*	(0)	(2)
3	:=	a	(2)

Nội dung

- Mã 3 địa chỉ
  - Các dạng mã,
  - Dịch trực tiếp cú pháp thành mã 3 địa chỉ
  - Cài đặt mã
- Sinh mã cho khai báo
  - Quy tắc ngữ nghĩa
  - Lưu trữ thông tin về phạm vi
- Sinh mã cho lệnh gán
  - Tên trong bảng ký hiệu
  - Địa chỉ hóa các phần tử của mảng
- Sinh mã cho các biểu thức logic
- Sinh mã cho các cấu trúc lập trình

Sinh mã cho khai báo

- Sử dụng biến toàn cục offset
  - Trước khi bắt đầu khai báo: offset = 0
  - Với mỗi khai báo biến sẽ đưa tên đối tượng, kiểu và giá trị của offset vào bảng ký hiệu
  - Tăng offset lên bằng kích thước của dữ liệu
- Các tên trong chương trình con được truy xuất thông qua địa chỉ tương đối offset
  - Khi gặp tên đối tượng (biến), dựa vào trường Offset để biết vị trí trong vùng dữ liệu

Sinh mã cho khai báo

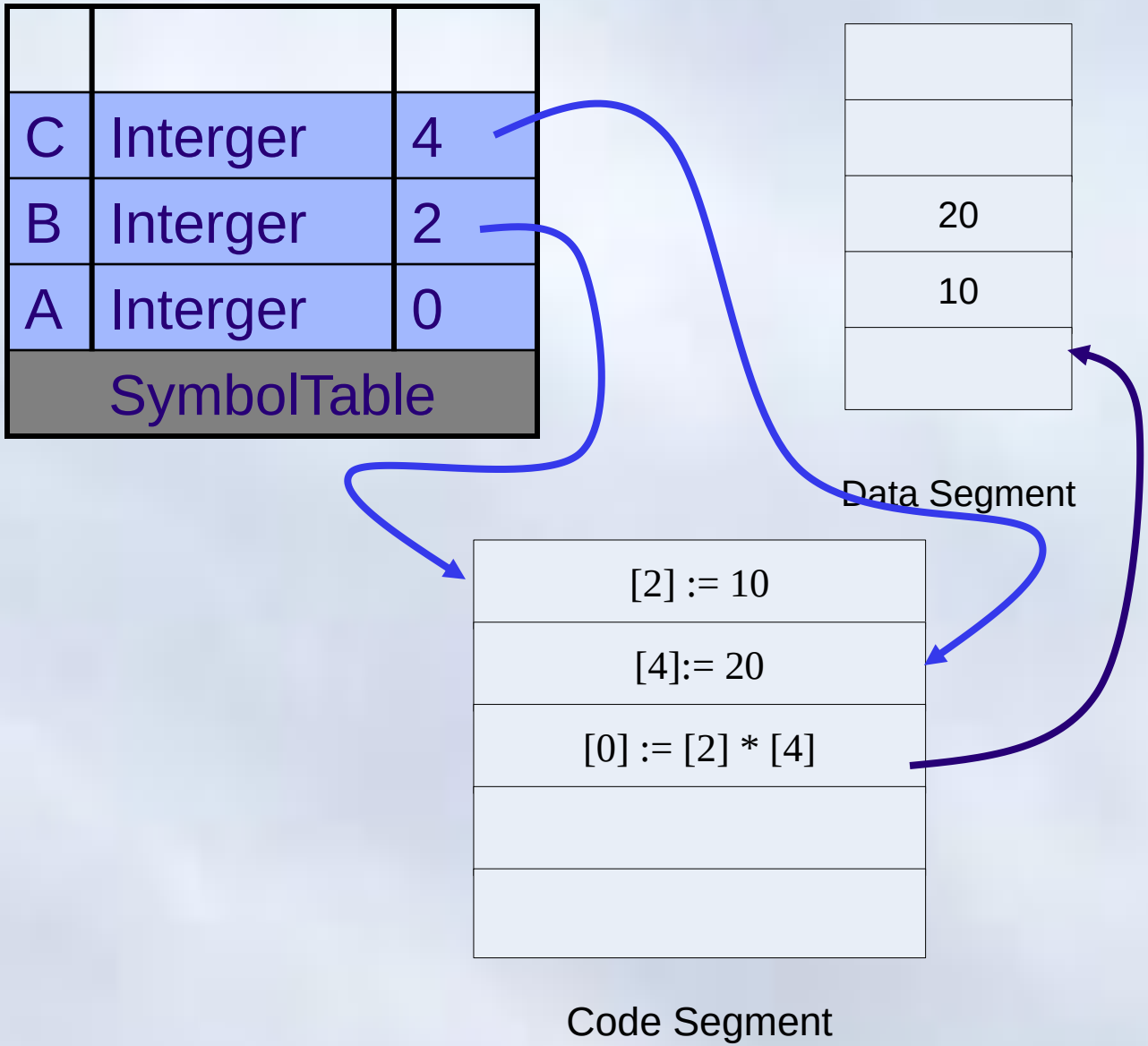
Sản xuất	Quy tắc ngữ nghĩa
P→ MD	{}
M → ε	{*Offset = 0*}
D → D ; D	
D → Id : T	*enter*(id.*name*, T.*type*, *offset*) *Offset =Offset +T.Width*
T → interger	T.*type = Interger; T.width = 2*
T → real	T.*type =real*; T.*width = 4*
T →array[num] of T_1	T.*type=array*(1..num.*val*,T_1.*type*) T.*width* = num.val * T_1.*width*

Hàm **Enter(name, type, offset)** thêm một đối tượng vào bảng ký hiệu với tên (name), kiểu(type) và địa chỉ tương đối (offset) của vùng dữ liệu của nó.

Sinh mã cho khai báo→Ví dụ

A: Interger;
B: Interger;
C:Interger;

B :=10
C:=20
A:=B*C



Lưu trữ thông tin về phạm vi

- Văn phạm cho phép các chương trình con bao nhau
  - Khi bắt đầu phân tích chương trình con, phần khai báo của chương trình bao tạm dừng
  - Dùng một bảng ký hiệu riêng co mỗi chương trình con
- Văn phạm của khai báo này:

  P → D

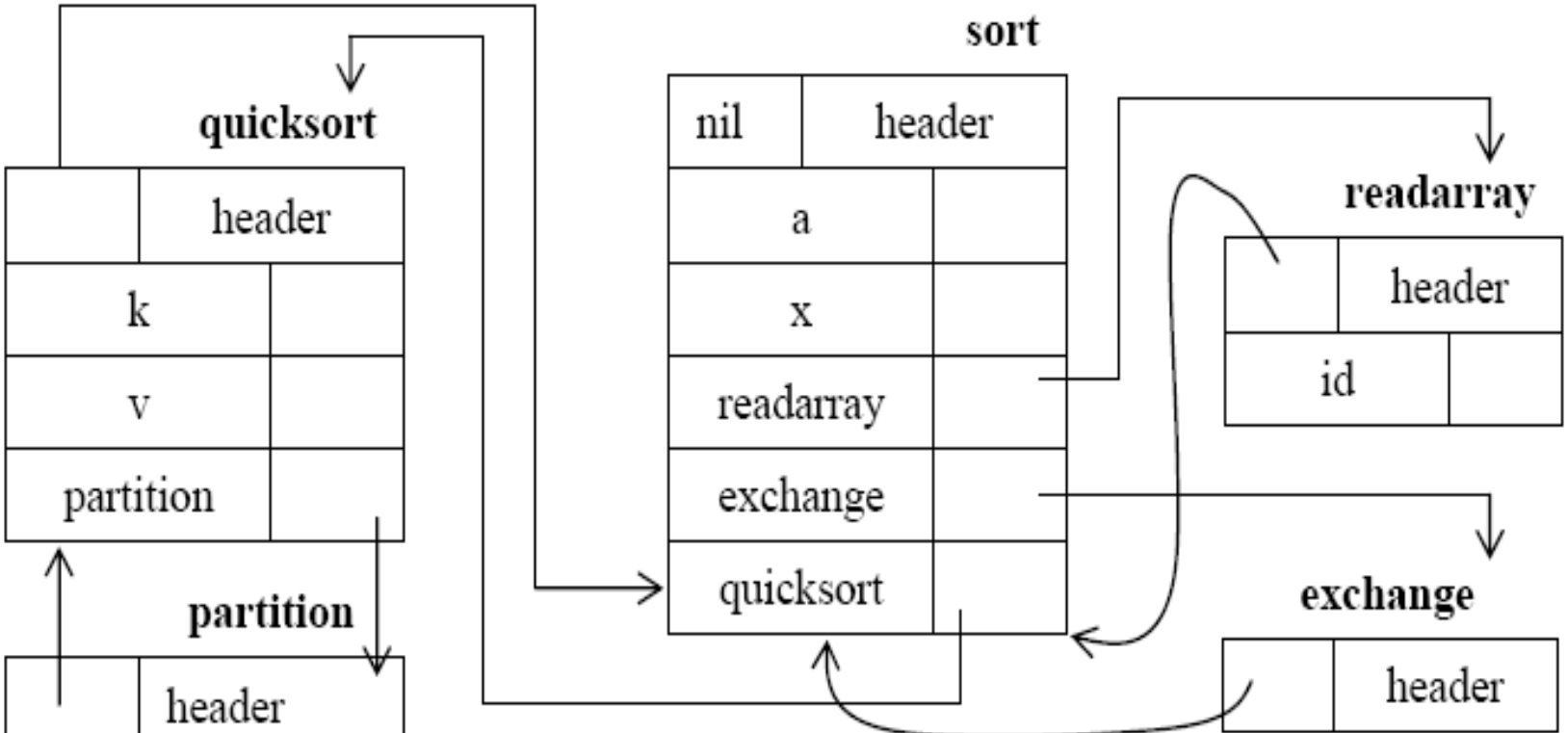
  D → D; D | id : T | proc id ; D ; S
- Khi khai báo chương trình con D → **proc** id D1; S được phân tích thì các khai báo trong D1 được lưu trong bảng kí hiệu mới.

Lưu trữ thông tin về phạm vi→Ví dụ

```
1) Program sort;      //Chương trình Quicksort
2) Var  a: array[0..10] of integer;
3)        x: integer;
4)      Procedure readarray;
5)        Var i: integer;
6)        Begin …… end {readarray};
7)     Procedure exchange(i, j: integer);
8)        Begin {exchange} end;
9)     Procedure quicksort(m, n: integer);
10)       Var k, v: integer;
11)    Function partition(y,z: integer): integer;
12)       Begin ……exchange(i,j) end; {partition}
13)    Begin … end; {quicksort}
14)Begin … end; {sort}
```

Lưu trữ thông tin về phạm vi→Ví dụ

- Các bảng ký hiệu của chương trình sort



Quy tắc ngữ nghĩa →Các thao tác

- **mktable(previous)** – tạo một bảng kí hiệu mới, bảng này có *previous* chỉ đến bảng cha của nó.
- **enter(table,name,type,offset)** – thêm một đối tượng mới có tên *name* vào bảng kí hiệu được chỉ ra bởi *table* và đặt kiểu là *type* và địa chỉ tương đối là *offset* vào các trường tương ứng.
- **enterproc(table,name,newbtable)** – tạo một phần tử mới trong bảng table cho chương trình con name, newtable trỏ tới bảng kí hiệu của CTC này.
- **addwidth(table,width)** – ghi tổng kích thước của tất cả các p/tử trong bảng kí hiệu vào header của bảng.

Khai báo trong chương trình con

Sản xuất	Quy tắc ngữ nghĩa
P→ MD	*addwidth*(*top*(tblptr), *top*(offset)); *pop*(tblptr); *pop*(offset)
M → ε	t:=*mktable*(null); *push*(t, tblptr); *push*(0, offset)
D → D ; D	
D →**proc id**; ND_1;S	t:=*top*(tblpr); *addwidth*(t,*top*(offset)); *pop*(tblptr); *pop*(offset); *enterproc*(*top*(tblptr), id.name, t)
N → ε	t:=*mktable*(*top*(tblptr)); *push*(t,tblptr); *push*(0,offset);
D → id : T	*enter*(*top*(tblptr),id.name,T.*type*, *top*(offset); *top*(offset):=*top*(offset) + T.*width*

Tblptr: là Stack dùng chứa các con trỏ trỏ tới bảng ký hiệu

Offset: Là Stack dùng lưu trữ các Offset

Xử lý các khai báo trong chương trình con

- Sản xuất: **P→MD** :
  - Hoạt động của cây con M được thực hiện trước
- Sản xuất: **M → ε:**
  - Tạo bảng ký hiệu cho phạm vi ngoài cùng (*chương trình sort*) bằng lệnh ***mktable(nil)*** ***//Không có SymTab cha***
  - Khởi tạo stack ***tblptr*** với bảng ký hiệu vừa tạo ra
  - Đặt offset = 0.
- Sản xuất: **N → ε:**
  - Tạo ra một bảng mới ***mktable(top(tblptr))***
    - Tham số ***top*(tblptr)** cho giá trị con trỏ tới bảng cha
  - Thêm bảng mới vào đỉnh stack **tblptr** //push(t,tblptr)
  - 0 được đẩy vào stack ***offset*** //push(0,Offset)

N đóng vai trò tương tự M khi một khai báo CTC xuất hiện

Xử lý các khai báo trong chương trình con

- Với mỗi khai báo **id: T**
  - một phần tử mới được tạo ra cho id trong bảng kí hiệu hiện hành (*top(tblptr)*)
  - Stack tblptr không đổi,
  - Giá trị **top(offset)** được tăng lên bởi T.width.
- Khi **D→ proc id ; N D_1 ; S** diễn ra
  - Kích thước của tất cả các đối tượng dữ liệu khai báo trong D_1 sẽ nằm trên đỉnh stack offset.
  - Kích thước này được lưu trữ bằng cách dùng Addwidth(),
  - Các stack tblptr và offset bị lấy mất phần tử trên cùng (pop())
  - Thao tác thực hiện trên các khai báo của chương trình con.

Nội dung

- Mã 3 địa chỉ
  - Các dạng mã,
  - Dịch trực tiếp cú pháp thành mã 3 địa chỉ
  - Cài đặt mã
- Sinh mã cho khai báo
  - Quy tắc ngữ nghĩa
  - Lưu trữ thông tin về phạm vi
- Sinh mã cho lệnh gán
  - Tên trong bảng ký hiệu
  - Địa chỉ hóa các phần tử của mảng
- Sinh mã cho các biểu thức logic
- Sinh mã cho các cấu trúc lập trình

Sinh mã cho lệnh gán → Các hàm

- Hàm lookup()
  - Tìm trong bảng kí hiệu xem một tên (*id.name*) đã tồn tại
    - Tìm trong bảng ký hiệu hiện thời (top(tblptr))
    - Nếu không có, tìm trong các bảng ký mức cha (*con trỏ trong phần header của bảng ký hiệu*)
  - Nếu tồn tại, trả về con trỏ tới vị trí; ngược lại, trả về nil.
- Thủ tục emit()
  - Ghi mã 3 địa chỉ vào một tập tin output
    - gen() xây dựng thuộc tính code cho các kí hiệu chưa kết thúc như
  - Khi thuộc tính code của kí hiệu không kết thúc trong vế trái sản xuất được tạo ra bằng cách nối thuộc tính code của kí hiệu không kết thúc trong vế phải theo đúng thứ tự xuất hiện, sẽ ghi ra tập tin bên ngoài

Sinh mã cho lệnh gán

Sản xuất	Quy tắc ngữ nghĩa
$S \rightarrow Id:=E$	p := lookup(id.name) if p<>nil then emit(p ':=' E.place) else error()
$E \rightarrow E_1+E_2$	E.*Place* = newTemp() emit(E.*place* ':=' E_1.*place* '+' E_2.*place*)
$E \rightarrow E_1*E_2$	E.*Place* = newTemp() emit(E.*place* ':=' E_1.*place* '*' E_2.*place*)
$E \rightarrow -E_1$	E.*place*= *newtemp()*; *emit*(E.*place* ':=' 'uminus' E_1.*place*)
$E \rightarrow (E)$	E.*place*= E_1.*place* ;
$E \rightarrow$ **Id**	p := lookup(id.name) if p<>nil then E.place := p else error()

Địa chỉ hóa các phần tử của mảng

- Các phần tử của mảng được lưu trữ trong một khối ô nhớ kế tiếp nhau để truy xuất nhanh
- **Mảng một chiều:** nếu kích thước một phần tử là w
  $\Rightarrow$ địa chỉ tương đối phần tử thứ ***i*** của mảng A là

$$\mathbf{A[i] = base + (i\text{-}low)*w}$$

$$A[i] = i*w + (base - low*w)$$

  - *Low*: cận dưới tập chỉ số. Một số ngôn ngữ, low = 0
  - *Base*: địa chỉ tương đối của ô nhớ cấp phát cho mảng(địa chỉ tương đối của phần tử A[low])
  - c = base – low*w có thể được tính tại thời gian dịch và lưu trong bảng kí hiệu. Vậy A[i] = i*w + c
- **Mảng 2 chiều:** mảng của mảng 1 chiều

Nội dung

- Mã 3 địa chỉ
  - Các dạng mã,
  - Dịch trực tiếp cú pháp thành mã 3 địa chỉ
  - Cài đặt mã
- Sinh mã cho khai báo
  - Quy tắc ngữ nghĩa
  - Lưu trữ thông tin về phạm vi
- Sinh mã cho lệnh gán
  - Tên trong bảng ký hiệu
  - Địa chỉ hóa các phần tử của mảng
- Sinh mã cho các biểu thức logic
- Sinh mã cho các cấu trúc lập trình

Sinh mã cho biểu thức logic

- Biểu thức logic được sỉnh bởi văn phạm sau:

  E→ E or E | E and E | not E

  | (E) | id relop id | true |false

- Trong đó:
  - **Or** và And kết hợp trái
  - Or có độ ưu tiên thấp nhất tiếp theo là And, và Not (*Văn phạm trên nhập nhằng*)
- Mã hóa giá trị logic true/false
  - Mã hóa bằng số; đánh giá một biểu thức logic như một biểu thức số học
  - Biểu diễn số **1: true, 0: false**

Sinh mã cho biểu thức logic→Ví dụ

- Biểu thức **a or b and not c**
  - Mã 3 địa chỉ:

    t1 = not c

    t2 = b and t1

    t3 = a or t2
- Biểu thức a < b
  - Tương đương lệnh **if a<b then 1 else 0**.
  - Mã 3 địa chỉ tương ứng (*g/thiết lệnh bắt đầu 100*)

    100: if a<b goto 103

    101: t:=0

    102: goto 104

    103: t:= 1

    104:

Sinh mã cho biểu thức logic

Sản xuất	Quy tắc ngữ nghĩa
$E \rightarrow E_1$ or E_2	E.Place =newTemp(); Emit(E.place '**:=**' E1.place '**or**' E2.place)
$E \rightarrow E_1$ and E_2	E.Place =newTemp(); Emit(E.place '**:=**' E1.place '**and**' E2.place)
$E \rightarrow$ not E_1	E.Place =newTemp(); Emit(E.place '**:=**' '**not**' E1.place)
$E \rightarrow Id_1$ relop Id_2	E.Place =newTemp(); Emit('**if**' id1.place relop id2.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**')
Nextstat (*next statement*) cho biết chỉ số của câu lệnh 3 địa chỉ tiếp theo	

Sinh mã cho biểu thức logic→Ví dụ

- Biểu thức a< b AND c > d
  - E→E and E → Id <Id and E → Id<Id and Id > Id

100: if a > b goto 103

101: t1 := 0

102: goto 104

103: t1 := 1

104: if c > d goto 106

105: t2 := 0

106: goto 108

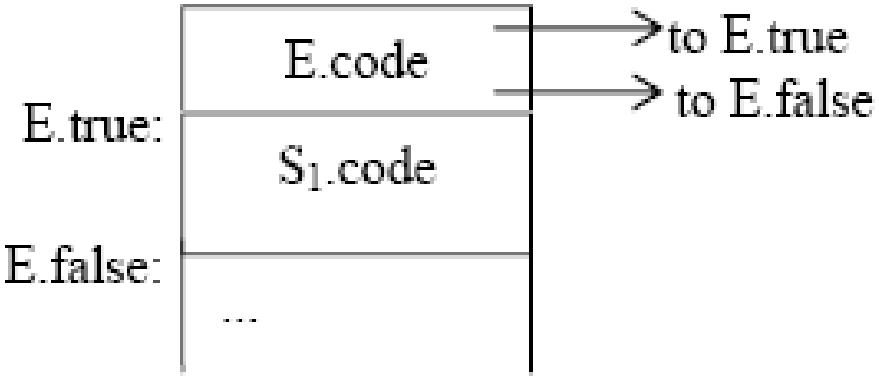
107: t2 := 1;

108: t3 := t1 and t2

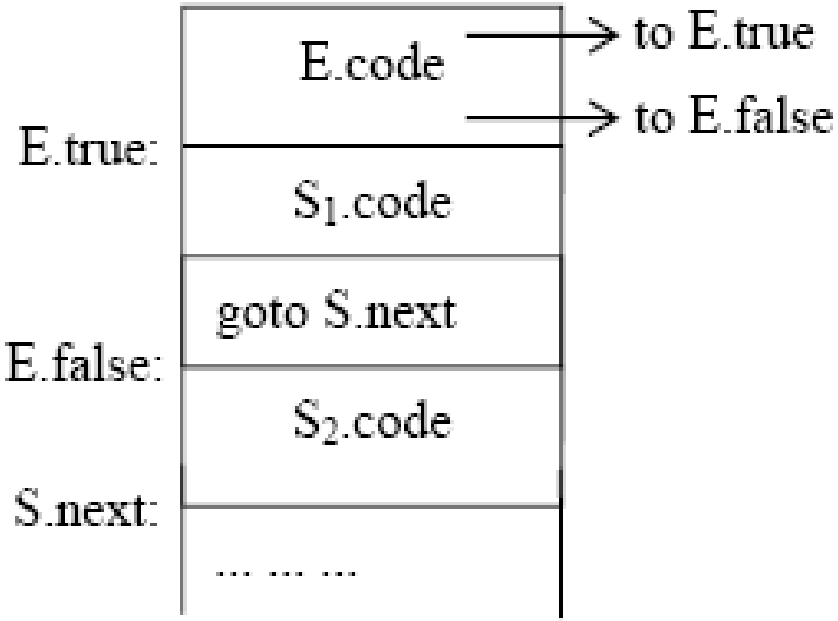
Sinh mã cho các cấu trúc lập trình

- Cấu trúc: S → if E then S_1 |
  if E then S_1 else S_2 |
  while E do S_1
- **E** là biểu thức logic. E có 2 nhãn
  - **E.true**: nhãn của dòng điều khiển nếu E là true
  - **E.false**: nhãn của dòng điều khiển nếu E là false
- **E.code**:mã lệnh 3 địa chỉ được sinh ra bởi S
- **S.next**: là nhãn mã lệnh 3 địa chỉ đầu tiên sẽ thực hiện sau mã lệnh của S
- **S.begin:** nhãn địa chỉ lệnh đầu tiên được sinh ra cho S

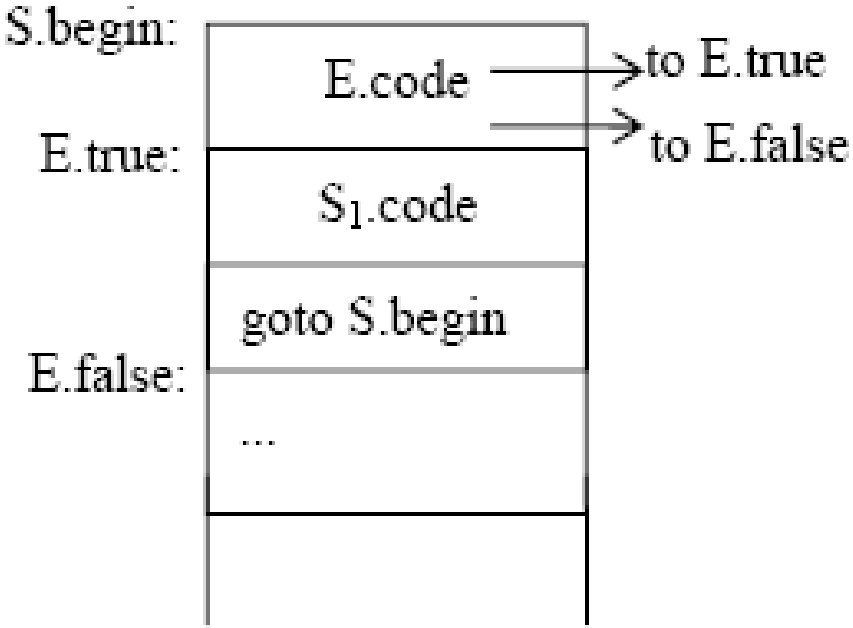
Sinh mã cho các cấu trúc lập trình



(a) if -then



(b) if -then-else



(c) while-do

Dịch trực tiếp cú pháp cho các cấu trúc lập trình

Sản xuất	Quy tắc 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 ' : ') || S1.code || gen('goto' S.next)|| gen(E.false ' : ') || S2.code
$S \rightarrow$ while E do S_1	S.Begin=newLabel(); E.True = newLabel(); E.False = S.next(); S_1.next = S.next S.Code = gen(S.begin ':') ||E.code ||gen(E.true ':') S_1.code || gen('**goto** ' S.Begin);

Sinh mã cho biểu thức logic trong cấu trúc lập trình

- Nếu E có dạng: a < b
  - Mã lệnh sinh ra có dạng

    If a<b then goto E.true

    else goto E.false

E.Code sinh ra như thế nào?

- Nếu E có dạng: E_1 or E_2 thì
  - Nếu E_1 là true thì E cũng là true
  - Nếu E_1 là false thì phải đánh giá E_2; E sẽ là true hay false phụ thuộc E2
- Tương tự với E1 and E2
- Nếu E có dạng not E_1
  - Nếu E_1 là true, E là false và ngược lại

Sinh mã cho biểu thức logic trong cấu trúc lập trình

Sản xuất	Quy tắc 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_1.false ‘: ‘) || E_2.code
$E \rightarrow E_1$ 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_1.true ‘: ‘) || E_2.code
Chú ý: E.True và E.false là các thuộc tính kế thừa	

Sinh mã cho biểu thức logic trong cấu trúc lập trình

Sản xuất	Quy tắc ngữ nghĩa
$E \rightarrow \text{not } E_1$	E_1.true := E.false E_1.false := E.true E.Code = E_1.code
$E \rightarrow (E_1)$	E_1.True := E.true E_1.false := E.false E.Code := E_1.code
$E \rightarrow Id_1$ relop Id_2	E.Code := gen('**if**' id1.place relop id2.place '**goto**' E.true) gen('goto' E.false);
E→True	E.Code := gen('goto' E.true)
E→False	E.Code := gen('goto' E.false)

Sinh mã cho biểu thức logic → Ví dụ

a < b or c < d and e < f

- Giả thiết Ltrue và Lfalse là nhãn đi đến ứng với các giá trị true và false của biểu thức
- Dựa trên quy tắc ngữ nghĩa, sinh ra

```
        if a < b goto Ltrue
        goto L1:
L1:     if c < d goto L2
        goto Lfalse
L2:     if e < f goto Ltrue
        goto Lfalse
```

Sinh mã cho biểu thức logic → Ví dụ

- **E→a < b**

  E.code = if a < b goto E.true goto E.false

- **E→E_1 or E_2**
  - E_1.true := E.true ⇒ E_1.true = Ltrue
    - Ltrue là nhãn đi tới nếu biểu thức là true
    - Lfalse là nhãn đi tới nếu biểu thức là false
  - E_1.false := L1;
  - E_2.true := E.true= Ltrue; E_2.false := E.false = Lfalse
  - E.Code = E_1.code || gen (E_1.false ‘: ‘) || E_2.code

    if a < b goto E_1.true goto E_1.false || E_1.false || E_2.code

    if a < b goto Ltrue goto L1 L1: || E_2.code (1)

Sinh mã cho biểu thức logic → Ví dụ

- **E→c < d** E.code = if c < d goto E.true goto E.false
- **E→e < f** E.code = if e < f goto E.true goto E.false
- **E→E_1 and E_2**
  - E_1.true := L2
  - E_1.false := E.false = LFalse;
  - E_2.true := E.true= Ltrue; E_2.false := E.false = Lfalse
  - E.Code = E_1.code || gen (E_1.true ‘: ‘) || E_2.code

  if c < d goto E_1.true goto E_1.false || E_1.true : || E_2.code

  if c<d goto L2 goto Lfalse L2: if e < f goto E_2.true goto E_2.false

  if c<d goto L2 goto Lfalse L2: if e < f goto Lttue goto Lfalse (2)

Biểu thức hỗn hợp

- Thực tế, các biểu thức logic thường chứa các biểu thức số học
  - (a+b)<c
- Xét văn phạm

  E →E + E | E and E | E relop E| Id

  E and E đòi hỏi 2 đối số phải là logic

  E + E, E relop E: Các đối số là biểu thức toán học
- Để sinh mã trong trường hợp phức hợp
  - Dùng thuôc tính tổng hợp E.Type cho biết kiểu là arith hay logic.

Chương 5: Sinh mã

1. Sinh mã trung gian
2. Sinh mã đích
3. Tối ưu mã

Nội dung

1. Chương trình đích
2. Máy ngăn xếp
   - Bộ lệnh
   - Bộ thông dịch cho máy ngăn xếp
3. Xây dựng bảng ký hiệu
   - Biến
   - Tham số
   - Hàm, thủ tục và chương trình
4. Sinh mã cho các lệnh cơ bản

1. Chương trình đích

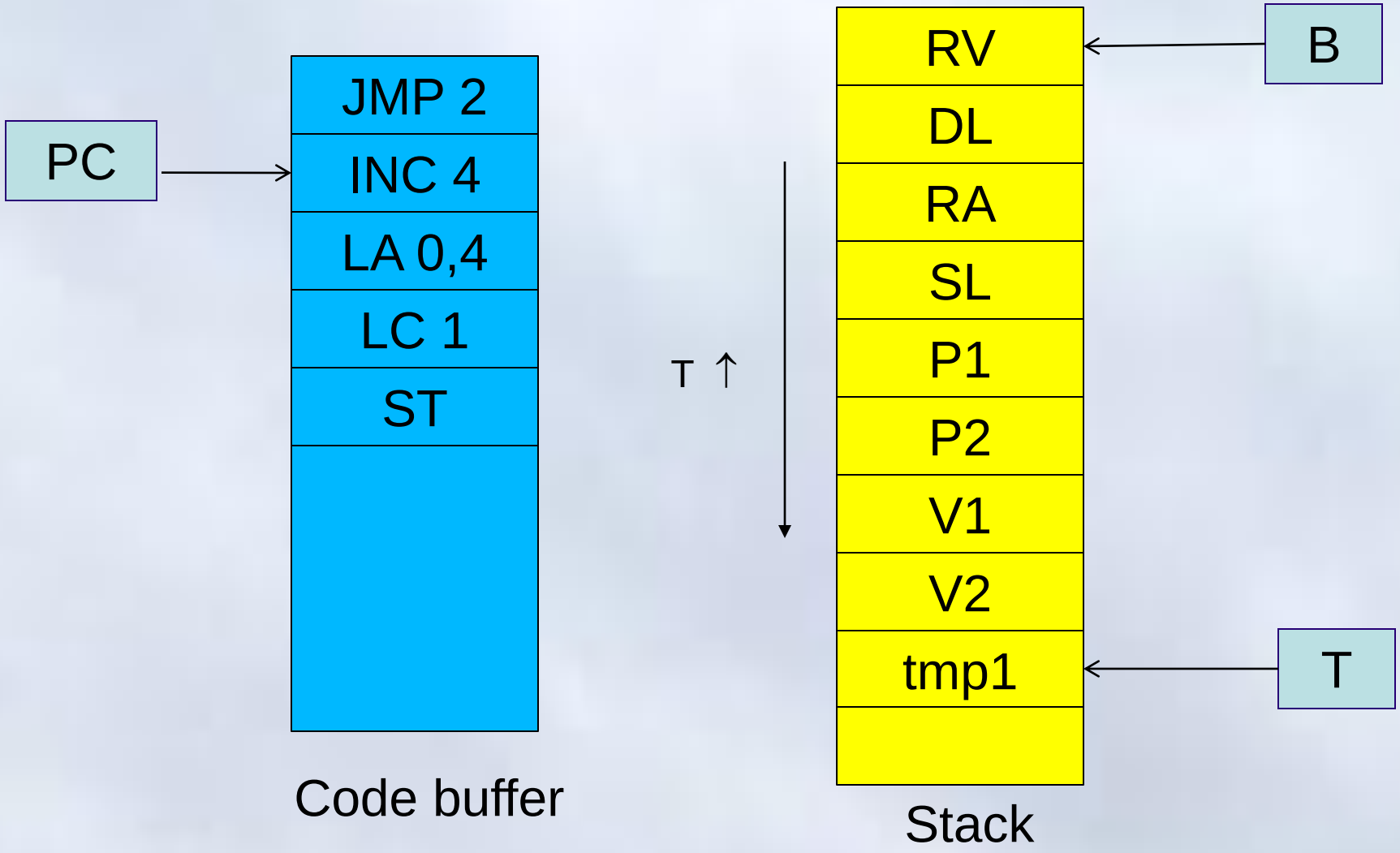
- Viết trên một ngôn ngữ trung gian
- Là dạng Assembly của máy giả định
  - Máy ảo
    - Máy ảo làm việc với bộ nhớ stack
- Việc thực hiện chương trình thông qua một bộ thông dich interpreter
  - Interpreter mô phỏng hành động của máy ảo
  - Thực hiện tập lệnh assembly của nó
- Chương trình đích được dịch từ
  - Mã nguồn
  - Mã trung gian

2. Máy ngăn xếp

- Máy ngăn xếp là một hệ thống tính toán
  - Sử dụng ngăn xếp để lưu trữ các kết quả trung gian của quá trình tính toán
  - Kiến trúc đơn giản
  - Bộ lệnh đơn giản
- Máy ngăn xếp có hai vùng bộ nhớ chính
  - **Khối lệnh:**
    - Chứa mã thực thi của chương trình
  - **Ngăn xếp:**
    - Lưu trữ các kết quả trung gian

2. Máy ngăn xếp

PC, B, T là các thanh ghi của máy



2. Máy ngăn xếp → Thanh ghi

- **PC (program counter):**
  - Con trỏ lệnh trỏ tới lệnh hiện tại đang thực thi trên bộ đệm chương trình
- **B (base):**
  - Con trỏ trỏ tới địa chỉ cơ sở của **vùng nhớ cục bộ**. Các biến cục bộ được truy xuất gián tiếp qua con trỏ này
- **T (top);**
  - Con trỏ, trỏ tới đỉnh của ngăn xếp

2. Máy ngăn xếp → Bản hoạt động

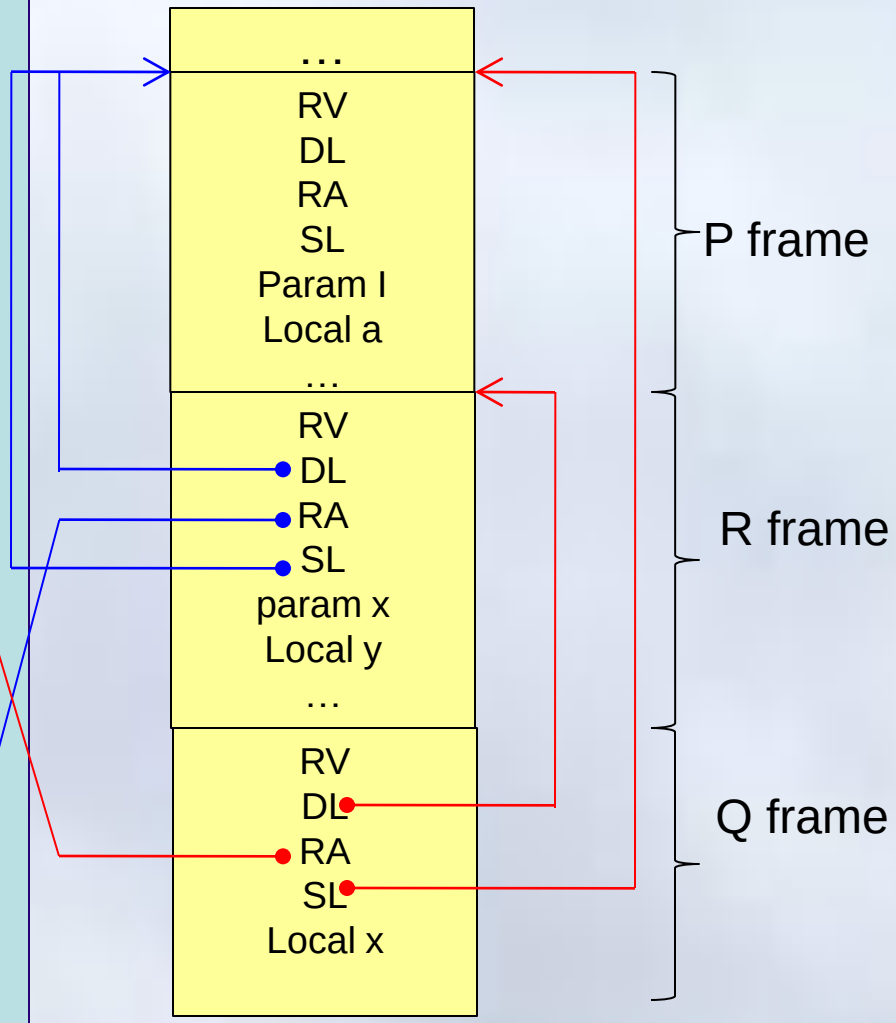
- Không gian nhớ cấp phát cho mỗi chương trình con (*hàm/thủ tục/chương trình chính*) khi chúng được kích hoạt
  - Lưu giá trị tham số
  - Lưu giá trị biến cục bộ
  - Lưu các thông tin quan trọng khác:
    - RV, DL, RA, SL
- Một chương trình con có thể có nhiều bản hoạt động

2. Máy ngăn xếp → Bản hoạt động (stack frame)

- RV (*return value*):
  - Lưu trữ giá trị trả về cho mỗi hàm
- DL (*dynamic link*):
  - Địa chỉ cơ sở của bản hoạt động của chương trình con gọi tới nó (caller).
  - Được sử dụng để hồi phục ngữ cảnh của chương trình gọi (caller) khi chương trình được gọi (called) kết thúc
- RA (*return address*):
  - Địa chỉ lệnh quay về khi kết thúc chương trình con
  - Sử dụng để tìm tới lệnh tiếp theo của caller khi called kết thúc
- SL (*static link*):
  - Địa chỉ cơ sở của bản hoạt động của chương trình con bao ngoài
  - Sử dụng để truy nhập các biến phi cục bộ

2. Máy ngăn xếp → Bản hoạt động → Ví dụ

```
Procedure P(I : integer);
  Var a : integer;
  Function Q;
    Var x : char;
    Begin
        …
      return
    End;
  Procedure R(X: integer);
    Var y : char;
    Begin
      …
      y = Call Q;
      …
    End;
  Begin
    …
    Call R(1);
    …
  End;
```

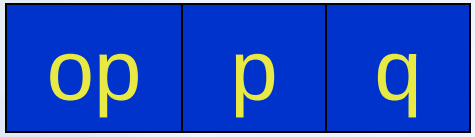


2. Máy ngăn xếp → Lệnh

- Lệnh máy có dạng : **Op p q**
  - Op : Mã lệnh
  - p, q : Các toán hạng.
    - Các toán hạng có thể tồn tại đầy đủ, có thể chỉ có 1 toán hạng, có thể không tồn tại
    - Ví dụ

```
J 1        % Nhảy đến địa chỉ 1
LA 0, 4    % Nạp địa chỉ từ số 0+4 lên đỉnh stack
HT         %Kết thúc chương trình
```

2. Máy ngăn xếp → Bộ lệnh (1/5)



LA	Load Address	t:=t+1; s[t]:=base(p)+q;
LV	Load Value	t:=t+1; s[t]:=s[base(p)+q];
LC	Load Constant	t:=t+1; s[t]:=q;
LI	Load Indirect	s[t]:=s[s[t]];
INT	Increment T	t:=t+q;
DCT	Decrement T	t:=t-q;

2. Máy ngăn xếp → Bộ lệnh (2/5)

op	p	q

J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];

2. Máy ngăn xếp → Bộ lệnh (3/5)

op	p	q

RC	Read Character	read one character into s[s[t]]; t:=t-1;
RI	Read Integer	read integer to s[s[t]]; t:=t-1;
WRC	Write Character	write one character from s[t]; t:=t-1;
WRI	Write Integer	write integer from s[t]; t:=t-1;
WLN	New Line	CR & LF

2. Máy ngăn xếp → Bộ lệnh (4/5)

op	p	q

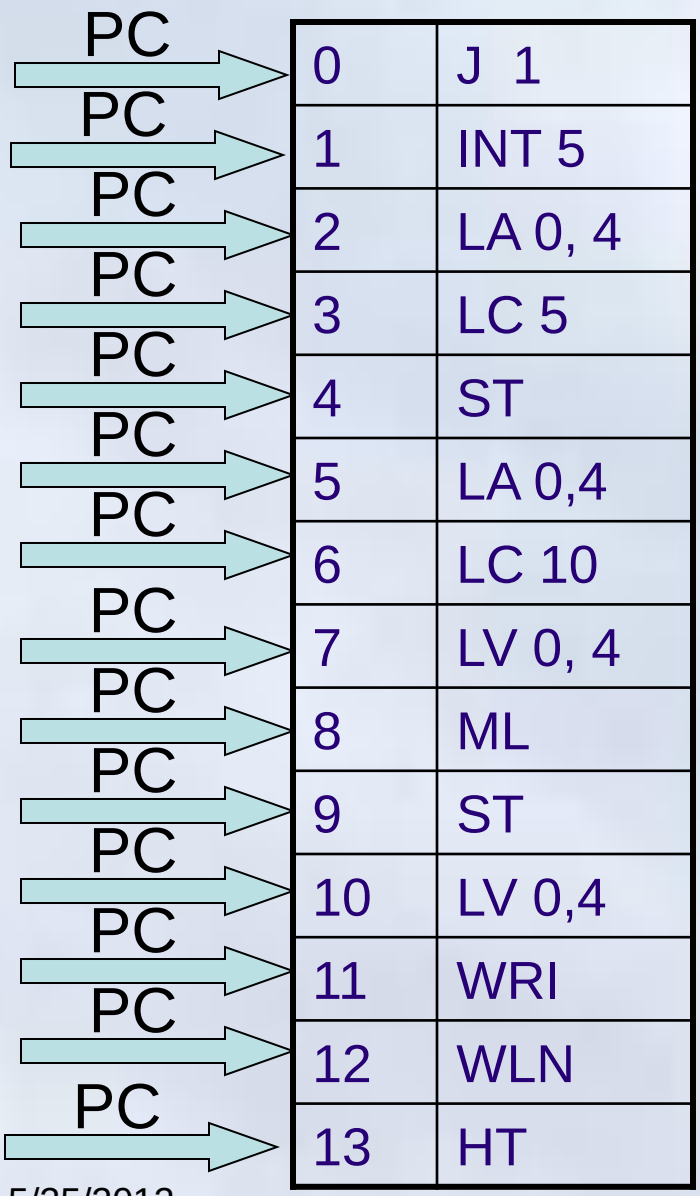
AD	Add	t:=t-1; s[t]:=s[t]+s[t+1];
SB	Subtract	t:=t-1; s[t]:=s[t]-s[t+1];
ML	Multiply	t:=t-1; s[t]:=s[t]*s[t+1];
DV	Divide	t:=t-1; s[t]:=s[t]/s[t+1];
NEG	Negative	s[t]:=-s[t];
CV	Copy Top of Stack	s[t+1]:=s[t]; t:=t+1;

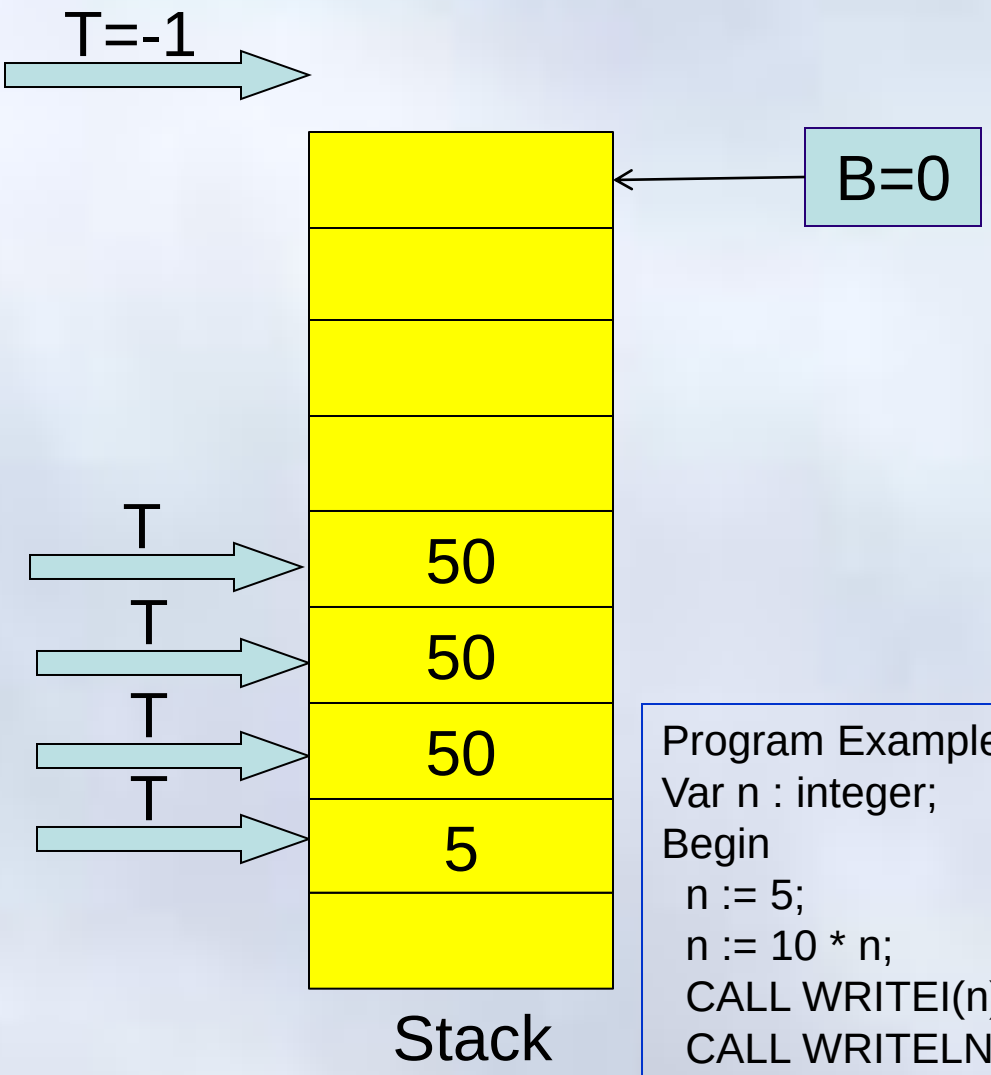
2. Máy ngăn xếp → Bộ lệnh (5/5)

op	p	q

EQ	Equal	t:=t-1; if s[t] = s[t+1] then s[t]:=1 else s[t]:=0;
NE	Not Equal	t:=t-1; if s[t] != s[t+1] then s[t]:=1 else s[t]:=0;
GT	Greater Than	t:=t-1; if s[t] > s[t+1] then s[t]:=1 else s[t]:=0;
LT	Less Than	t:=t-1; if s[t] < s[t+1] then s[t]:=1 else s[t]:=0;
GE	Greater or Equal	t:=t-1; if s[t] >= s[t+1] then s[t]:=1 else s[t]:=0;
LE	Less or Equal	t:=t-1; if s[t] <= s[t+1] then s[t]:=1 else s[t]:=0;

2. Máy ngăn xếp → Ví dụ






```
Program Example1;
Var n : integer;
Begin
  n := 5;
  n := 10 * n;
  CALL WRITEI(n);
  CALL WRITELN;
End.
```

2. Máy ngăn xếp → Bộ thông dịch cho máy ngăn xếp

- kplrun (Cài đặt trong thực hành)
  - $ kplrun <source> [-s=stack-size] [-c=code-size] [-debug] [-dump]
- Các tham số dòng lệnh
  - s: định nghĩa kích thước stack
  - c: định nghĩa kích thước tối đa của mã nguồn
  - dump: In mã ASM
  - debug: chế độ gỡ rối

2. Máy ngăn xếp → Bộ thông dịch cho máy ngăn xếp

Instructions.c

```
enum OpCode {
  OP_LA,    // Load Address:
  OP_LV,    // Load Value:
  OP_LC,    // load Constant
  OP_LI,    // Load Indirect
  OP_INT,   // Increment t
  OP_DCT,   // Decrement t
  OP_J,     // Jump
  OP_FJ,    // False Jump
  OP_HL,    // Halt
  OP_ST,    // Store
  OP_CALL,  // Call
  OP_EP,    // Exit Procedure
  OP_EF,    // Exit Function
```

```
  OP_RC,    // Read Char
  OP_RI,    // Read Integer
  OP_WRC,   // Write Char
  OP_WRI,   // Write Int
  OP_WLN,   // WriteLN
  OP_AD,    // Add
  OP_SB,    // Substract
  OP_ML,    // Multiple
  OP_DV,    // Divide
  OP_NEG,   // Negative
  OP_CV,    // Copy Top
  OP_EQ,    // Equal
  OP_NE,    // Not Equal
  OP_GT,    // Greater
  OP_LT,    // Less
  OP_GE,    // Greater or Equal
  OP_LE,    // Less or Equal
  OP_BP     // Break point.
};
```

Máy ngăn xếp → Bộ thông dịch cho máy ngăn xếp

Instructions.c

```
struct Instruction_ {
  enum OpCode op;
  WORD p;
  WORD q;
};


struct CodeBlock_ {
  Instruction* code;
  int codeSize;
  int maxSize;
};
```

```
CodeBlock* createCodeBlock(int maxSize);
void freeCodeBlock(CodeBlock* codeBlock);
void printInstruction(Instruction* instruction);
void printCodeBlock(CodeBlock* codeBlock);

void loadCode(CodeBlock* codeBlock, FILE* f);
void saveCode(CodeBlock* codeBlock, FILE* f);

int emitLA(CodeBlock* codeBlock, WORD p, WORD q);
int emitLV(CodeBlock* codeBlock, WORD p, WORD q);
int emitLC(CodeBlock* codeBlock, WORD q);
….
int emitLT(CodeBlock* codeBlock);
int emitGE(CodeBlock* codeBlock);
int emitLE(CodeBlock* codeBlock);

int emitBP(CodeBlock* codeBlock);
```

2. Máy ngăn xếp → Bộ thông dịch cho máy ngăn xếp

gencode.c

```
void initCodeBuffer(void);
void printCodeBuffer(void);
void cleanCodeBuffer(void);
int serialize(char* fileName);

int genLA(int level, int offset);
int genLV(int level, int offset);
int genLC(WORD constant);
….
int genLT(void);
int emitGE(void);
int emitLE(void);
```

3. Xây dựng bảng ký hiệu

- Bổ sung thông tin cho biến
  - Vị trí trên frame
  - Phạm vi
- Bổ sung thông tin cho tham số
  - Vị trí trên frame
  - Phạm vi
- Bổ sung thông tin cho hàm/thủ tục/chương trình
  - Địa chỉ bắt đầu
  - Kích thước của frame
  - Số lượng tham số của hàm/thủ tục

Bổ sung thông tin cho biến

- Vị trí trên frame của biến
  - Vị trí tính từ base của frame
- Phạm vi

```
struct VariableAttributes_ {
   Type *type;
   struct Scope_ *scope;
   int localOffset;
};
```

Bổ sung thông tin cho tham số

- Vị trí trên frame của tham số
  - Vị trí tính từ base của frame
- Phạm vi

```
struct ParameterAttributes_ {
  enum ParamKind kind;
  Type* type;
  struct Scope_ *scope;
  int localOffset;
};
```

Bổ sung thông tin cho phạm vi

- Kích thước của frame

```
struct Scope_ {
   ObjectNode *objList;
   Object *owner;
   struct Scope_ *outer;
   int frameSize;
};
```

Bổ sung thông tin cho thủ tục

- Vị trí
- Số lượng tham số

```
struct ProcedureAttributes_ {
   struct ObjectNode_ *paramList;
   struct Scope_* scope;

   int paramCount;
   CodeAddress codeAddress;
};
```

Bổ sung thông tin cho hàm

- Vị trí
- Số lượng tham số

```
struct FunctionAttributes_ {
   struct ObjectNode_ *paramList;
   Type* returnType;
   struct Scope_ *scope;

   int paramCount;
   CodeAddress codeAddress;
};
```

Bổ sung thông tin cho chương trình

- Vị trí

```
struct ProgramAttributes_ {
   struct Scope_ *scope;
   CodeAddress codeAddress;
};
```

Hàm int sizeOfType(type* t)

- Trả về số ngăn nhớ trên stack mà một biến thuộc kiểu của tham số truyền vào sẽ chiếm.
  - Nếu kiểu của t là TP_INT/TP_CHAR
    - return INT_SIZE/CHAR_SIZE
      - *Theo quy ước, kiểu integer/char đều chiếm 1 từ trên stack*
  - Nếu kiểu của t là TP_ARRAY
    - return arraySize * sizeOfType(elementType)

void declareObject(Object* obj)

- Nếu là đối tượng toàn cục (currentScope=NULL)
  - Đưa vào symtab->GlobalObjectList
- Đối tượng khác:
  - Đưa vào symtab->currentScope->objList
  - Xét các trường hợp khác nhau của đối tượng được khai báo
    - Variable
    - Hàm
    - Thủ tục
    - Tham số

void declareObject(Object* obj)

- Đối tượng khác là **Variable**:
  - Cập nhật scope = currentScope
  - Cập nhật

    localOffset = currentScope -> frameSize
  - Tăng kích thước frameSize
    - frameSize += sizeOfType(Obj->varAttrs->type)

void declareObject(Object* obj)

- Đối tượng cục bộ là **Function**
  - Cập nhật outer = currentScope
- Đối tượng cục bộ là **Procedure**
  - Cập nhật outer = currentScope

void declareObject(Object* obj)

- Đối tượng cục bộ là **Parameter**
  - Cập nhật scope = currentScope
  - Cập nhật

    localOffset = currentScope->frameSize
  - Tăng kích thước frameSize
  - Cập nhật paramList của owner
  - Tăng paramCount của owner.

4.Sinh mã cho các lệnh

- **Lệnh gán** V := Exp

```
<code of Lvalue v>// đẩy địa chỉ của v lên stack
<code of exp>      // đẩy giá trị của exp lên stack
ST                 //S[S[t-1]] = S[t]
```

4.Sinh mã cho các lệnh→ **Lệnh rẽ nhánh**

If <dk> Then statement

```
    <code of dk>     // đẩy giá trị điều kiện dk lên stack
    FJ  L             //Nhảy có điều kiện, L = 0
    <code of statement> //Sinh mã cho statement
L:      //Cập nhật lại nhãn L bằng kúc thước đoạn mã
        //được sinh ra bởi statement
```

If <dk> Then st1 Else st2

```
    <code of dk> // đẩy giá trị điều kiện dk lên stack
    FJ L1       //L1 hiện giờ bằng 0
    <code of st1>
    J L2        //L2 = 0
L1:             //Cập nhật lại nhãn cho L1
    <code of st2>
L2:             //Cập nhật lại nhãn cho L2
```

4.Sinh mã cho các lệnh→ **Lệnh lặp while**

While <dk> Do statement

```
L1:     <code of dk> // Nhãn L1 xác định
        FJ L2           //genFJ(0)
        <code of statement>
        J L1
L2:             //Cập nhật giá trị cho nhãn L2
```

```
begin = getCurentCodeAddress()//Xác định L1
compileCondition()                  //sinh mã cho Condition
Jmp = genFJ(0)      //Sinh ra mã lệnh nhay tơi dia chi 0
compileStatement()
GenJ(begin)
Jmp.q = getCurrentCodeAddress()//sua la dia chi nhay toi
```

4.Sinh mã cho các lệnh→ Lệnh lặp for

For v := exp1 to exp2 do statement

```
    <code of Lvalue v>              //Đặt địa chỉ v lên stack
    genCV   // Sinh mã CV: nhân đôi địa chỉ của v =Copy địa chỉ V lên stack
    <code of exp1>                  //Đặt giá trị của biểu thức exp1 lên stack
    genST()   // Sinh mã lệnh ST: lưu giá trị của exp1 vào giá trị cua v
L1:                 //Sinh đia chi cho nhan L1: getCurrentCodeAddress()
    genCV()
    genLI()       // lấy giá trị của v & lưu vào đỉnh stack. Đỉnh stack là giá trị v
    <code of exp2>//Sinh mã để đặt giá trị của exp2 lên stack
    genLE ()      //Lệnh LE để so sánh 2 giá trị trên đỉnh stack: v và Exp2
    genFJ(L2)     //sinh mã lệnh FJ, nhay tới nhãn L2 chưa xác định
    <code of statement> //Sinh mã lệnh cho phần segment
    genCV() || genCV();  //Đỉnh satck là địa chỉ của v, Sao chép 2 lần
    genLI();             //Đinh stack là giá trị của v
    genLC(1) || genAD(); //Đặt lên stack 1 rồi cộng với giá trị của v
    genST();             // Lưu giá trị mới của v (v+1) vào vị trí của v
    genJ( L1)            //nhảy tới nhãn L1 đã tính
L2:                      //Tính toán nhãn L2 và cập nhật L2 tương ứng
    DCT 1                //Giảm thanh ghi T1 đơn vị do đỉnh stack là v
```

Lấy địa chỉ/giá trị biến

- Khi lấy địa chỉ/giá trị một biến cần tính đến phạm vi của biến
  - Biến cục bộ được lấy từ frame hiện tại
  - Biến phi cục bộ được lấy theo các StaticLink với cấp độ lấy theo "độ sâu" của phạm vi hiện tại so với phạm vi hiện thời
    - Hàm **computeNestedLevel(Scope* scope)** trả về độ sâu của biến

```
Level = 0;
Scope* tmp = symtab->currentScope;
While (tmp != scope) {tmp = tmp ->outer, level++ };
```

Lấy địa chỉ của tham số hình thức

- Khi **`LValue`** là tham số
- Cũng cần tính độ sâu như biến
  - Nếu là tham trị:
    - Địa chỉ cần lấy chính là địa chỉ của tham trị
  - Nếu là tham biến:
    - Giá trị của tham biến chính là địa chỉ muốn truy nhập nên địa chỉ cần lấy chính là giá trị của tham biến.

Lấy giá trị của tham số hình thức

- Khi tính toán giá trị của **`Factor`**
- Cũng cần tính độ sâu như biến
  - Nếu là tham trị: giá trị của tham trị chính là giá trị cần lấy.
  - Nếu là tham biến: giá trị của tham số là địa chỉ của giá trị cần lấy.

Lấy địa chỉ của giá trị trả về của hàm

- Giá trị trả về luôn nằm ở offset 0 trên frame
- Chỉ cần tính độ sâu giống như với biến hay tham số hình thức

Sinh lời gọi hàm/thủ tục

- Lời gọi
  - Hàm gặp trong sinh mã cho **factor**
  - Thủ tục gặp trong sinh mã lệnh **CallSt**
- Trước khi sinh lời gọi hàm/thủ tục cần phải nạp giá trị cho các tham số hình thức bằng cách
  - Tăng giá trị T lên 4 (bỏ qua RV,DL,RA,SL)
  - Sinh mã cho k tham số thực tế
  - Giảm giá trị T đi 4 + k
  - Sinh lệnh CALL

Sinh mã cho lệnh CALL (p, q)

Giả sử cần sinh lệnh CALL cho hàm/thủ tục A

Lệnh CALL có hai tham số:

- p: Độ sâu của lệnh CALL, chứa static link.
  Base(p) = base của frame chương trình con chứa khai báo của A.
- q: Địa chỉ lệnh mới
  q + 1 = địa chỉ đầu tiên của dãy lệnh cần thực hiện khi gọi A.

CALL (p, q)

```
s[t+2]:=b;          // Lưu lại dynamic link
s[t+3]:=pc;         // Lưu lại return address
s[t+4]:=base(p);    // Lưu lại static link
b:=t+1;       // Base mới và return value
pc:=q;              // địa chỉ lệnh mới
```

Hoạt đôngh khi thực hiện lệnh CALL (p, q)

- Điều khiển **pc** chuyển đến địa chỉ bắt đầu của chương trình con /* pc = p */
- **pc** tăng thêm 1 /* pc ++ */
- Lệnh đầu tiên thông thường là lệnh nhảy **J** để bỏ qua mã lệnh của các khai báo hàm/ thủ tục cục bộ trên *code buffer*.
- Lệnh tiếp theo là lệnh **INT** tăng **T** đúng bằng kích thước *frame* để bỏ qua *frame* chứa vùng nhớ của các tham số và biến cục bộ.

Hoạt đôngh khi thực hiện lệnh CALL (p, q)

- Thực hiện các lệnh và stack biến đổi tương ứng.
- Khi kết thúc
  - Thủ tục (lệnh **EP**): toàn bộ frame được giải phóng, con trỏ **T** đặt lên đỉnh frame cũ.
  - Hàm (lệnh **EF**): frame được giải phóng, chỉ chừa giá trị trả về tại offset 0, con trỏ **T** đặt lên đầu frame hiện thời (offset 0).

Chương 5: Sinh mã

1. Sinh mã trung gian
2. Sinh mã đích
3. Tối ưu mã

Giới thiệu

Yêu cầu

- Chương trình sau khi tối ưu phải tương đương
- Tốc độ thực hiện trung bình tăng
- Hiệu quả đạt được tương xứng với công sức

Có thể tối ưu mã vào lúc nào

- Mã nguồn
  - Do người lập trình (giải thuật)
- Mã trung gian
- Mã đích

Tối ưu mã cục bộ

- Nguyên tắc
  - Xem xét một dãy lệnh trong mã đích và thay thế chúng bằng những đoạn mã ngắn hơn và hiệu quả hơn
- Xu hướng chính
  - Loại bỏ lệnh dư thừa
  - Thông tin dòng điều khiển
  - Loại bỏ biểu thức con chung
  - Tính toán giá trị hằng
  - Giảm chi phí tính toán
  - Lan truyền biến gán…..

Loại bỏ lệnh dư thừa

- **Mã không đến được**

```
goto L2
x := x + 1        ← Không cần
L2:…
```

- **Mã chết**

```
x := 32
y := x + y
```

Nếu x không được dùng trong những lệnh tiếp theo, có thể chuyển thành:

```
y := 32 + y
```

Thông tin dòng điều khiển

goto L1

...

L1: goto L2 ← Không cần

chuyển thành

goto L2

Loại bỏ biểu thức con chung

- Ví dụ câu lệnh a[i+1] = b[i+1]
- Sinh ra mã

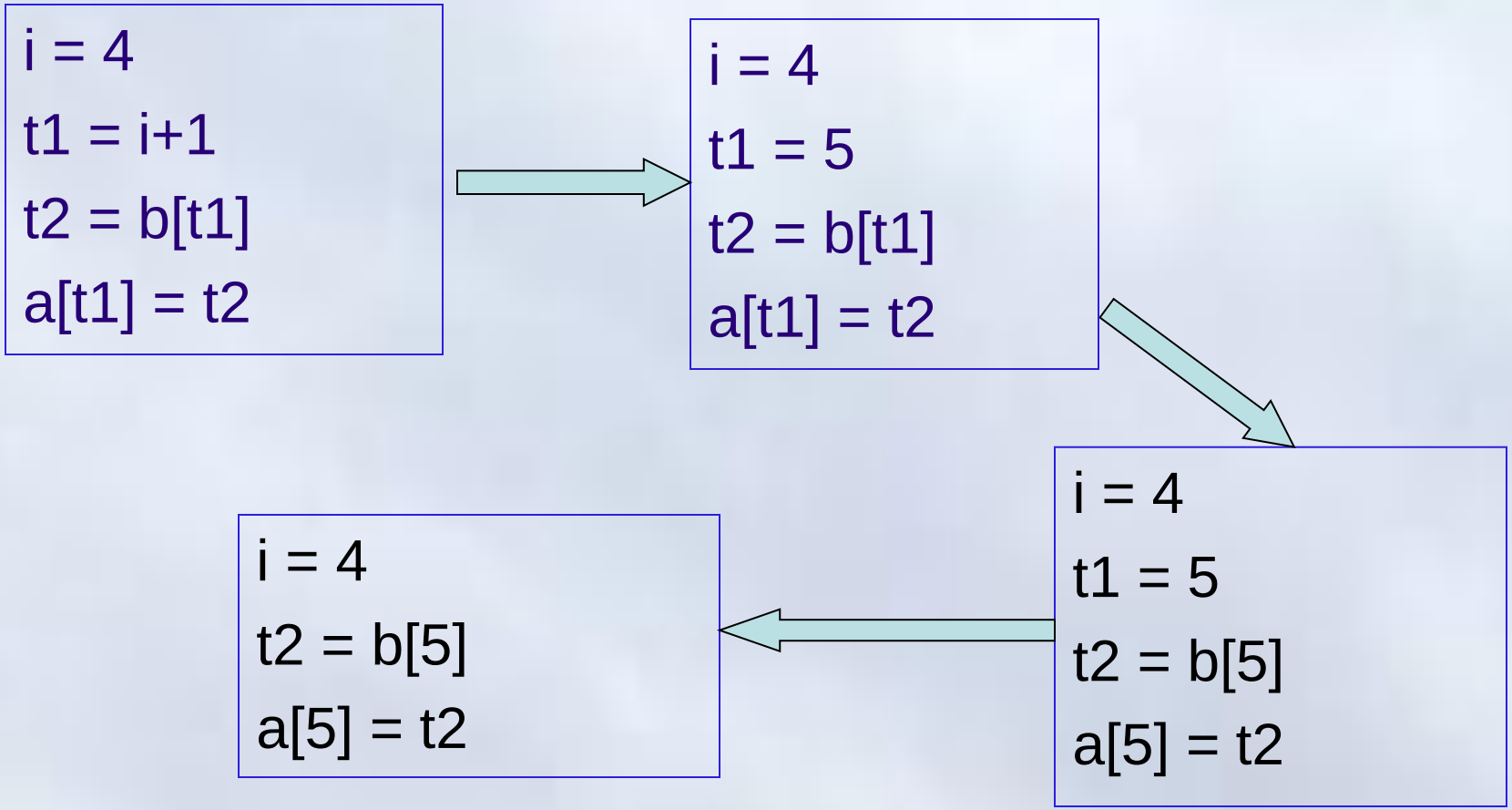
```
t1 = i+1
t2 = b[t1]
t3 = 4*t1
a[t3] = t2
a[t3] = t2
```

Không cần

Tính giá trị hằng

x := 32 trở thành x := 64

x := x + 32



Giảm chi phí tính toán

- **Tối ưu vòng lặp**

  Chuyển những đoạn mã bất biến ra ngoài vòng lặp:

  while (i <= limit - 2)

  ⇒

  t := limit - 2

  while (i <= t)

- **Tối ưu toán học**

  x := x + 0 ← Không cần

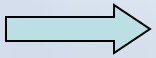
  A := sqrt(x) → A := x * x

  x := x * 2 → x := x + x

  x := x * 8 → x := x << 3

Lan truyền biến gán

```
tmp2 = tmp1 ;
tmp3 = tmp2 * tmp1;
tmp4 = tmp3 ;
tmp5 = tmp3 * tmp2 ;
c = tmp5 + tmp4 ;
```



```
tmp3 = tmp1 * tmp1;
tmp5 = tmp3 * tmp1 ;
c = tmp3 + tmp5 ;
```

Khối cơ bản (basic block)

- Khái niệm
  - 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

```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t2
t5 := b * b
t6 := t4 + t5
```

Giải thuật phân chia các khối cơ bản

- **Input:**
  - Dãy lệnh ba địa chỉ.
- **Output:**
  - DS các khối cơ bản với mã ba địa chỉ của từng khối
- **Phương pháp:**
  - Xác định tập các lệnh đầu, của từng khối cơ bản
    i. Lệnh đầu tiên của chương trình là lệnh đầu.
    ii. Bất kỳ lệnh nào là đích nhảy đến của các lệnh GOTO có hoặc không có điều kiện là lệnh đầu
    iii. Bất kỳ lệnh nào đi sau lệnh GOTO có hoặc không có điều kiện là lệnh đầu
  - Với mỗi lệnh đầu, khối cơ bản bao gồm nó và tất cả các lệnh tiếp theo không phải là lệnh đầu hay lệnh kết thúc chương trình

Giải thuật phân chia các khối cơ bản →Ví dụ

1. prod := 0
2. i := 1
3. t1 := 4 * i
4. t2 := a[t1]
5. t3 := 4 * i
6. t4 := b[t3]
7. t5 := t2 * t4
8. t6 := prod + t5
9. prod := t6
10. t7 := i + 1
11. i := t7
12. if i<=20 goto 3

- Lệnh (1) là lệnh đầu theo quy tắc i,
- Lệnh (3) là lệnh đầu theo quy tắc ii
- Lệnh sau lệnh (12) là lệnh đầu theo quy tắc iii.
- Các lệnh (1)và (2) tạo nên khối cơ bản thứ nhất.
- Lệnh (3) đến (12) tạo nên khối cơ bản thứ hai.

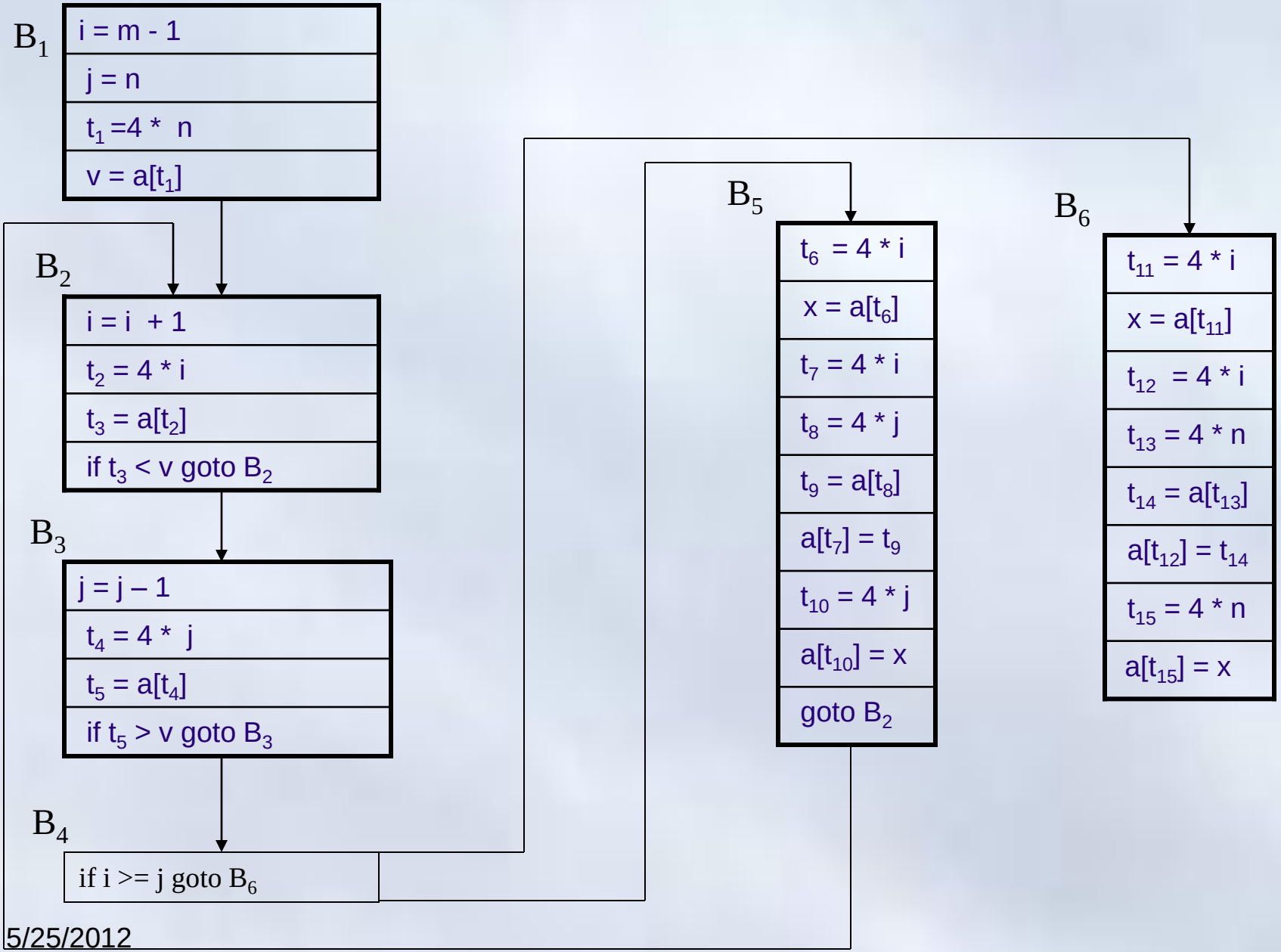
Mã ba địa chỉ của Quick Sort

1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]

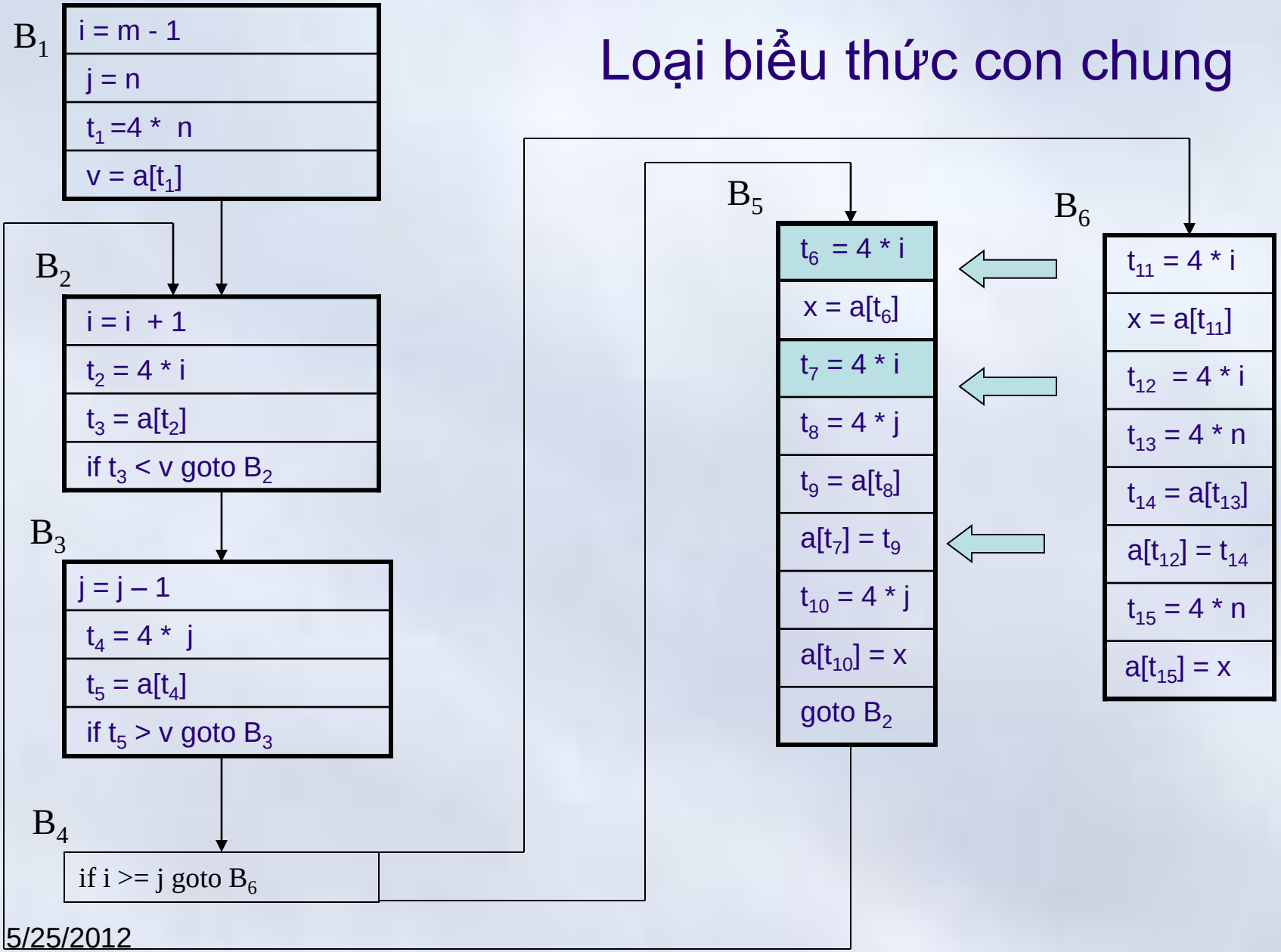
Xác định khối cơ bản

16	t_7 = 4 * I
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * I
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

Đồ thị quan hệ các khối

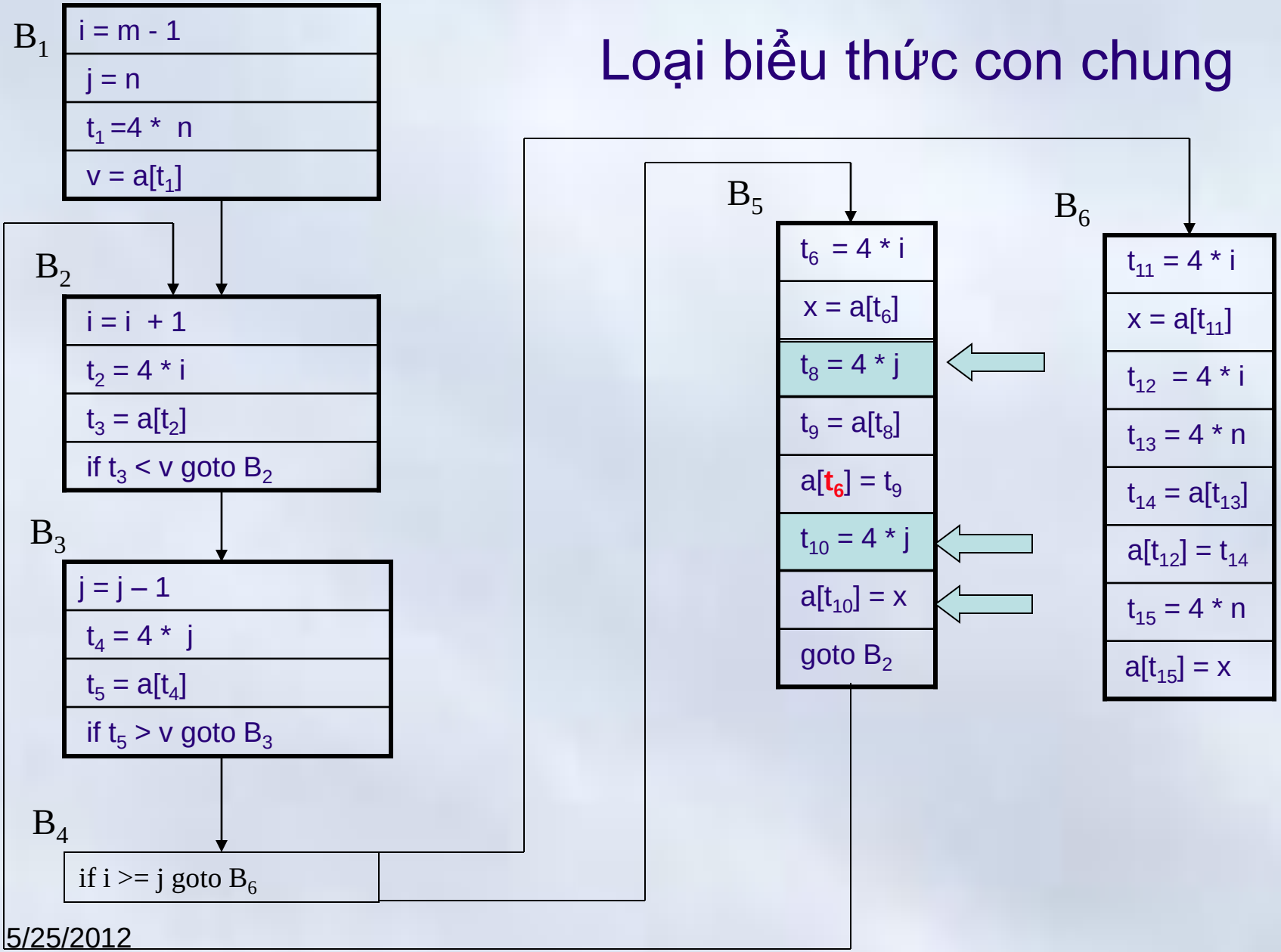


Tối ưu mã → Ví dụ



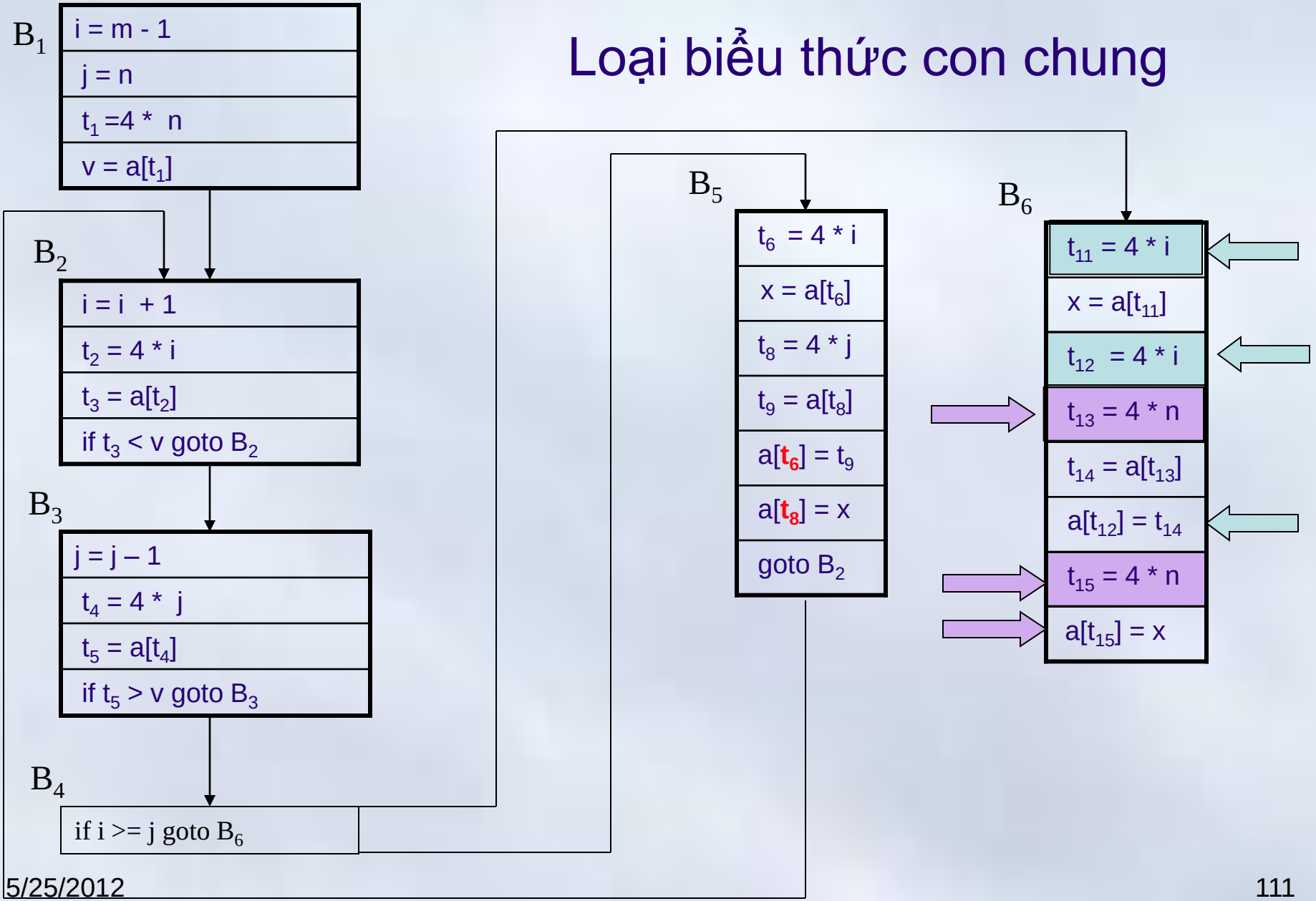
Loại biểu thức con chung

Đồ thị quan hệ các khối

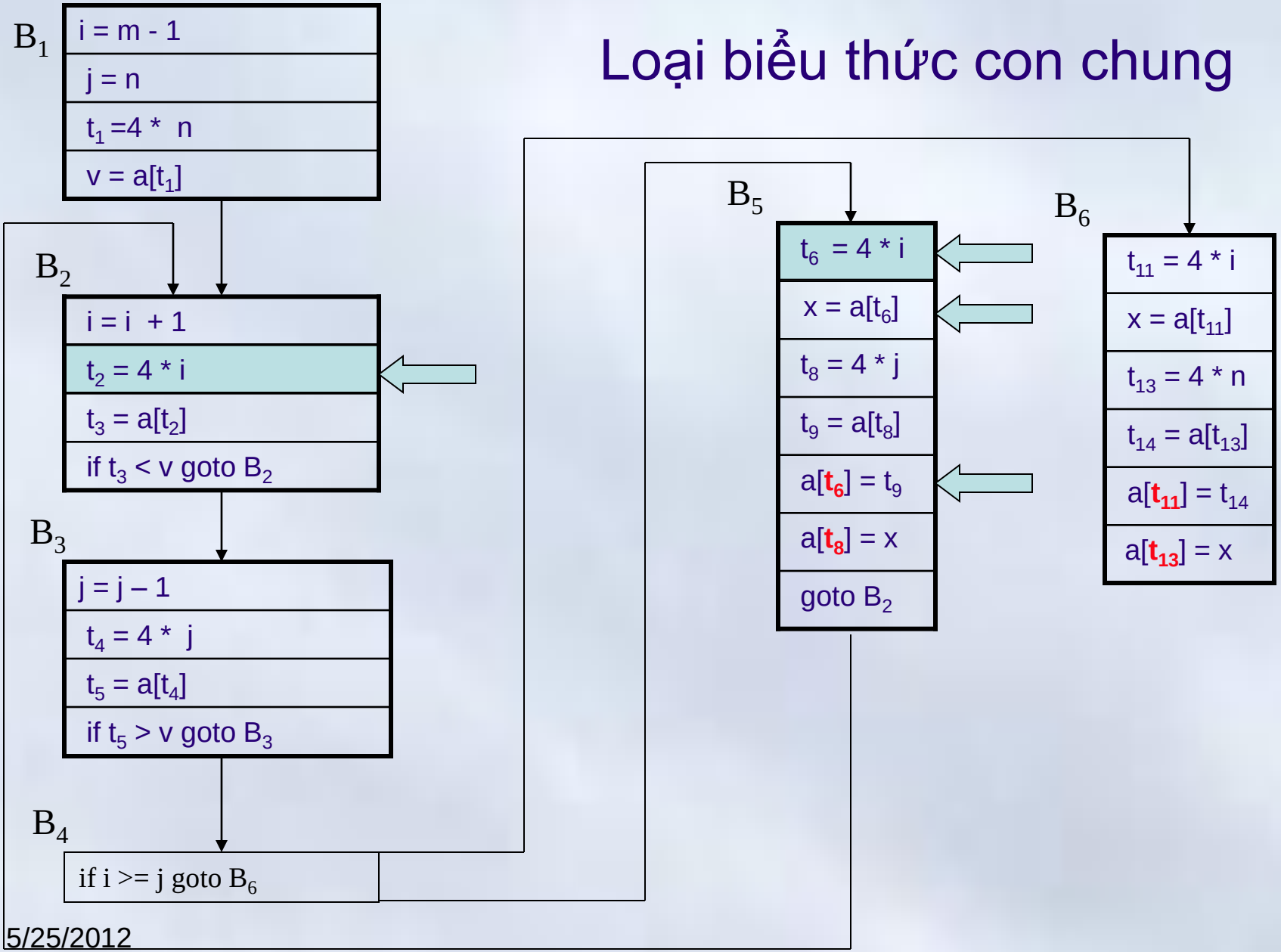


Loại biểu thức con chung

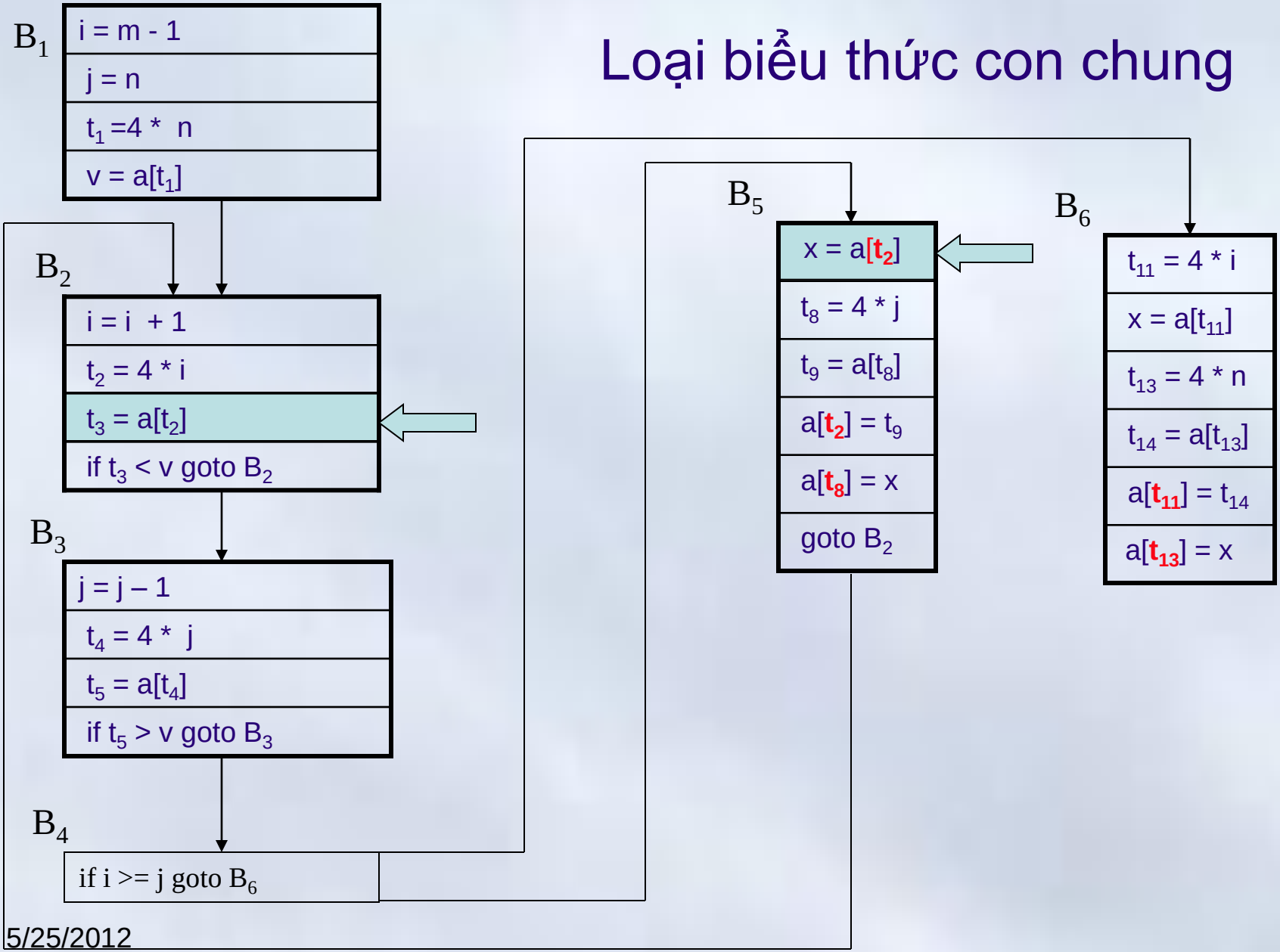
Tối ưu mã → Ví dụ



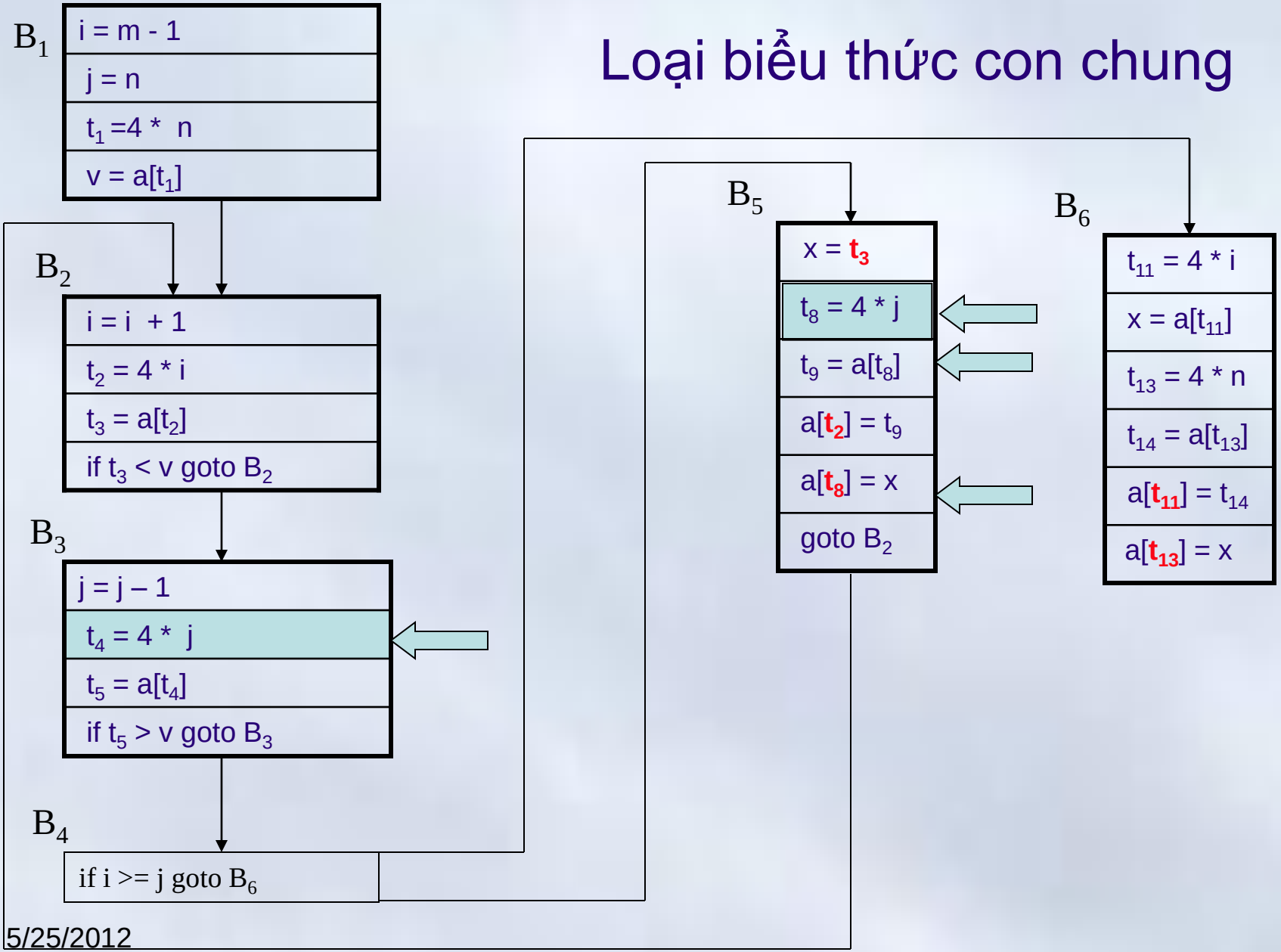
Tối ưu mã → Ví dụ



Tối ưu mã → Ví dụ

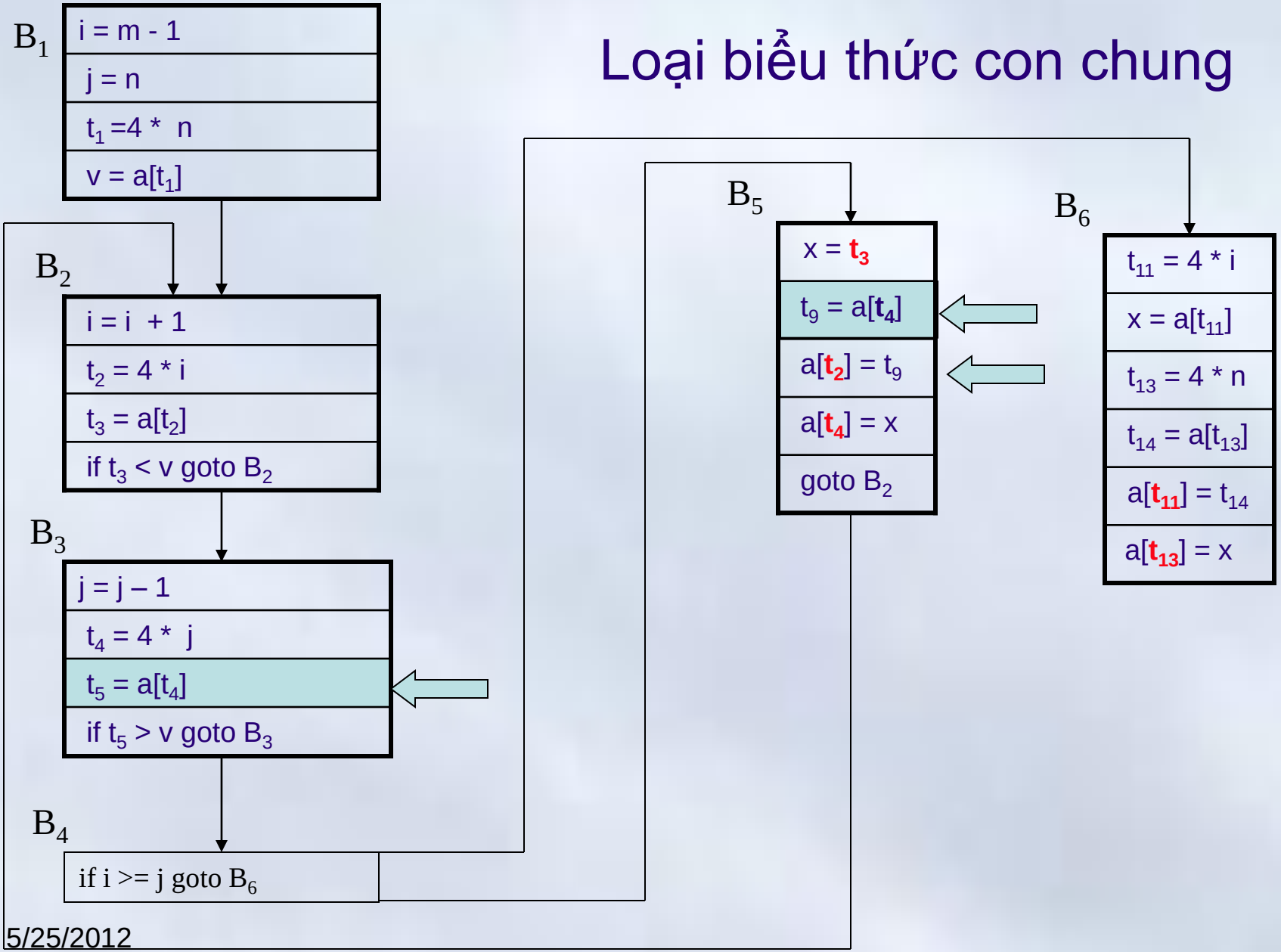


Tối ưu mã → Ví dụ



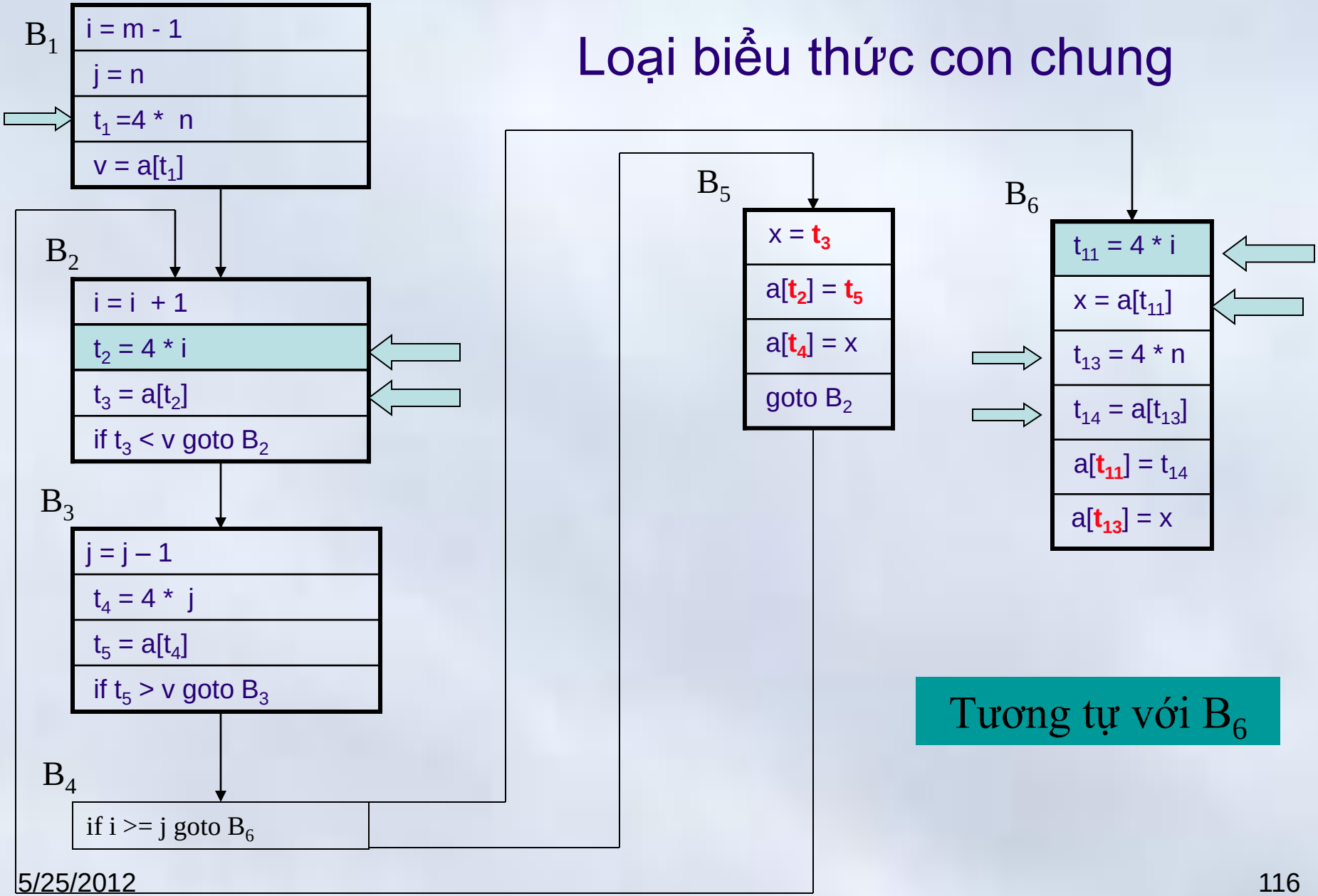
Loại biểu thức con chung

Tối ưu mã → Ví dụ

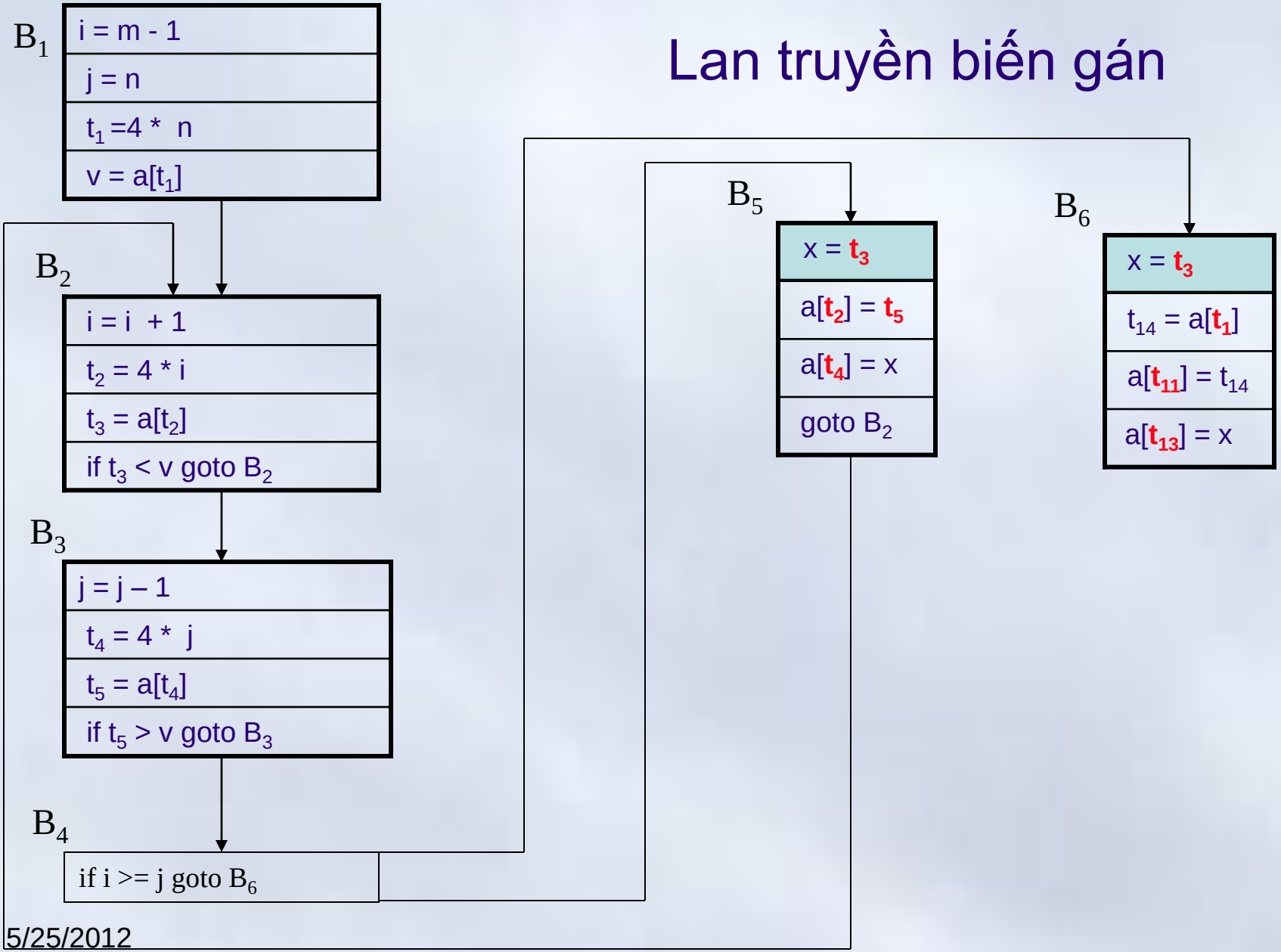


Loại biểu thức con chung

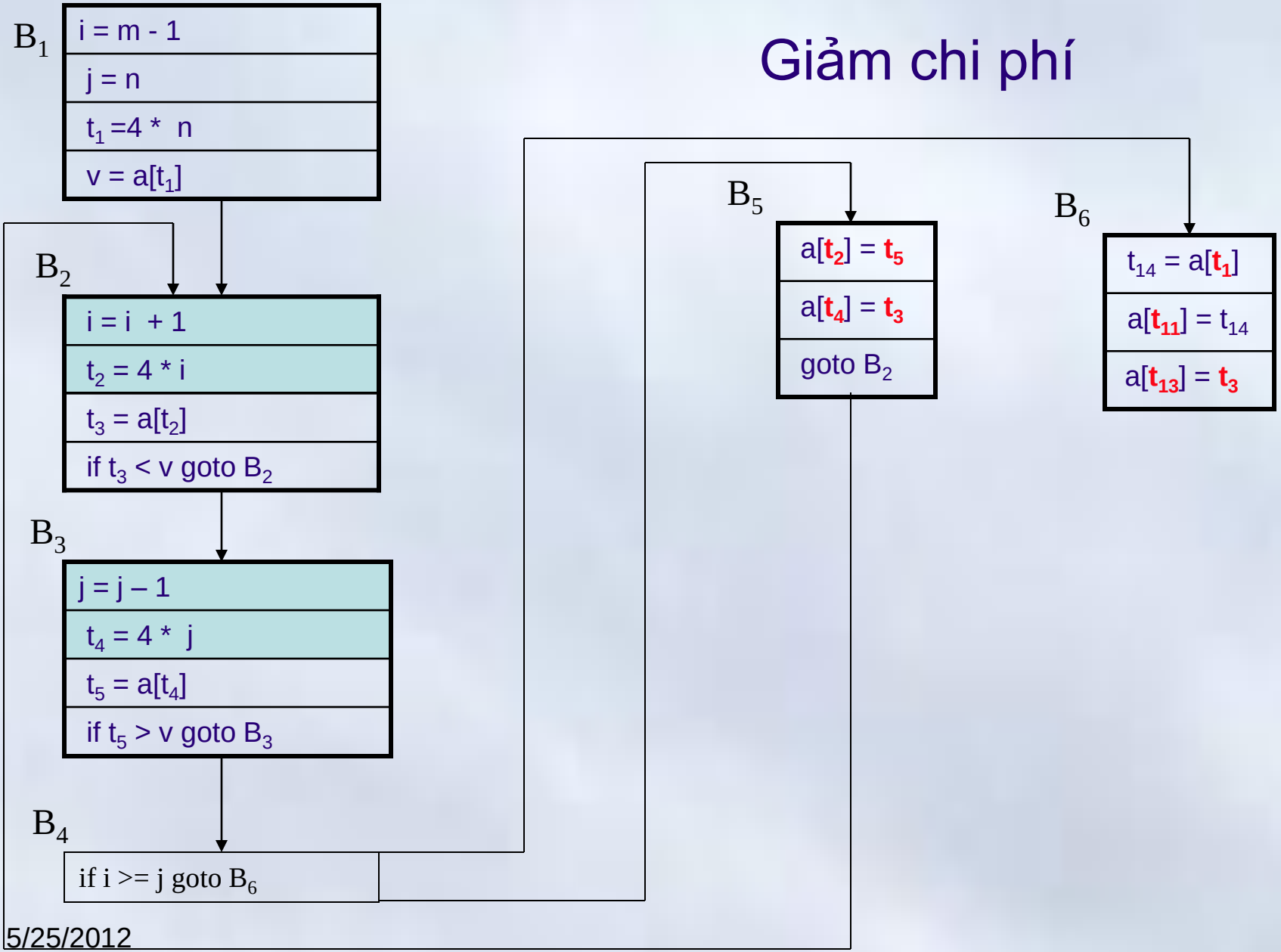
Đồ thị quan hệ các khối



Tối ưu mã → Ví dụ

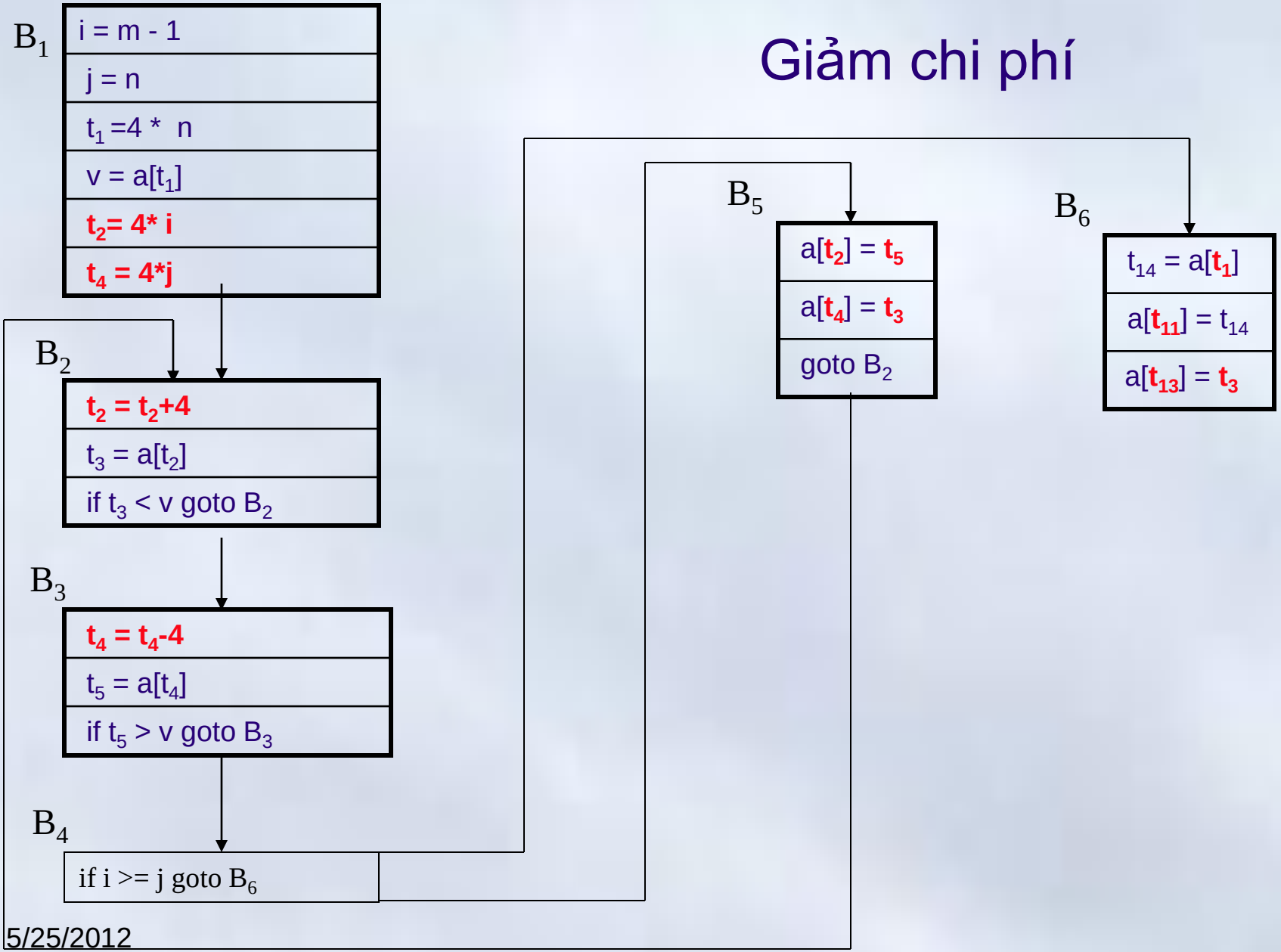


Tối ưu mã → Ví dụ



Giảm chi phí

Tối ưu mã → Ví dụ



Giảm chi phí

Tối ưu mã đích

- Không tồn tại một giải thuật tổng quát để tìm ra chương trình tối ưu nhất
  - Dùng một số thuật toán đơn giản để tối ưu mã đích
- Xét ví dụ máy có 1 thanh ghi **R** và tồn tại các lệnh
  - **Load m** → Nạp vào thanh ghi R giá trị m (R=m)
  - **Add m** → Cộng vào thanh ghi R giá trị m (R +=m)
  - **Mul m** → Nhân vào thanh ghi R giá trị m (R *=m)
  - **Store m** → Ghi giá trị thanh ghi R vào địa chỉ m

  Trong đó, m có thể là hằng số, biến

Tối ưu mã đích→Phép biến đổi đơn giản

- Phép giao hoán
  - Hai lệnh liên tiếp **Load** α và **Add** β có thể được thay thế bằng **Load** β và **Add** α
  - Hai lệnh liên tiếp **Load** α và **Mul** β có thể được thay thế bằng **Load** β và **Mul** α
- Dãy lệnh **store** α **Load** α sẽ bị hủy bỏ nếu
  - α không được sử dụng hoặc được lưu giá trị mới trước khi dùng (store α)
- Dãy lệnh **Load** α **store** β sẽ bị hủy bỏ nếu
  - Sau đó là lệnh load khác và không có sự thay đổi β từ đó về sau, đồng thời sự sử dung α được thay bằng β

Tối ưu mã đích→Phép biến đổi đơn giản →Ví dụ

- Load c
- Store #3
- Load b
- Store #1
- Load a
- Add #1
- Store #2
- Load 10
- Mul #2
- Add # 3
- Store V

- Load c
- Stroe #3
- Load b
- Store #1
- Load # 1
- Add a
- Strore #2
- Load #2
- Mul 10
- Add # 3
- Store V

- Load c
- Stroe #3
- Load b
- Add a
- Mul 10
- Add # 3
- Store V

- Load b
- Add a
- Mul 10
- Add c
- Store V