编译原理与设计：IR

北京理工大学 计算机学院

中间代码生成：概览

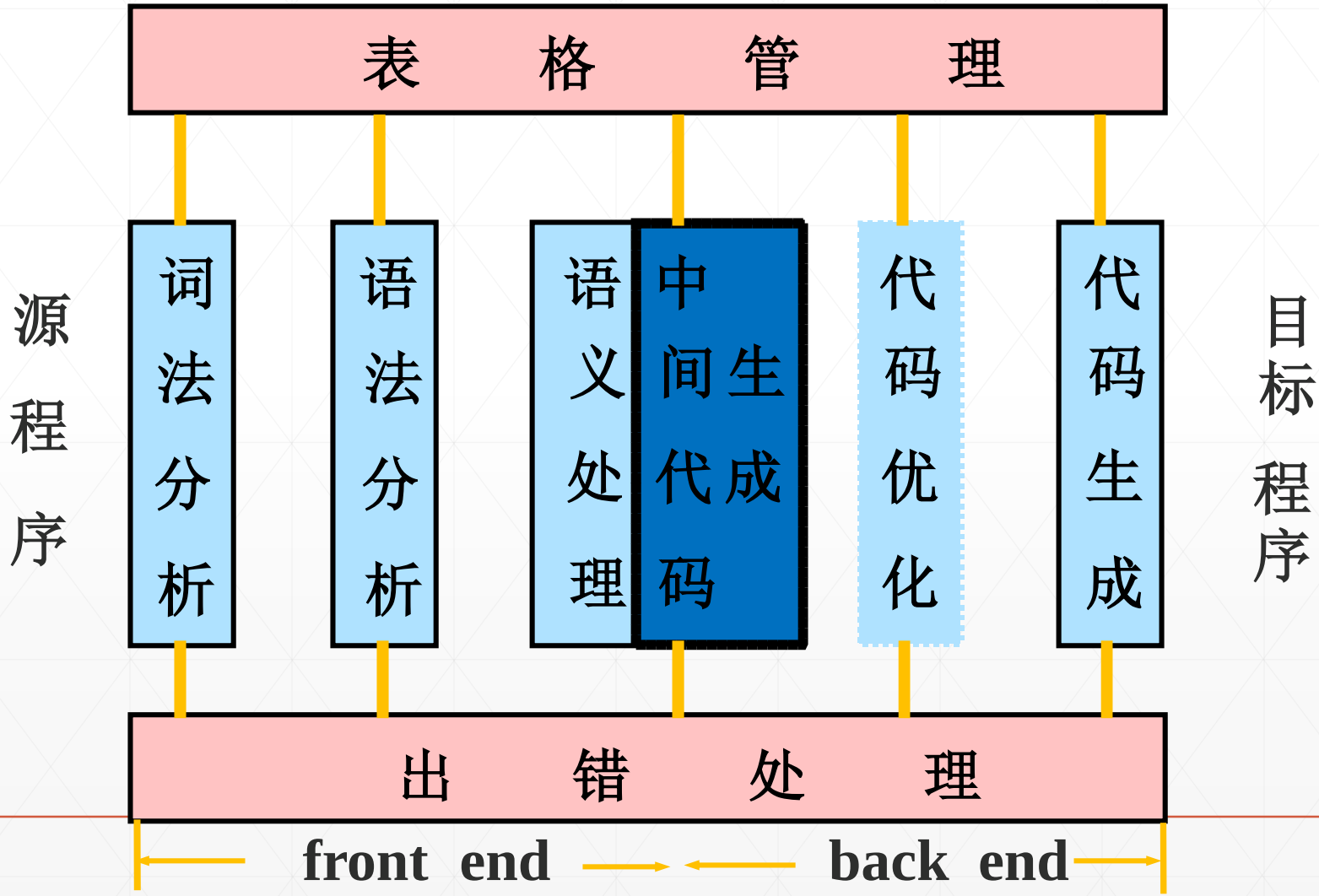


表 格 管 理
源程序
词法分析
语法分析
语义处理
中间代码生成
代码优化
代码生成
目标程序
出 错 处 理
front end
back end

语句翻译设计要点

- 确定语句的目标结构

源语句 ⇨ 中间及目标代码的布局

语义 ↑

If e_r then S_1 →

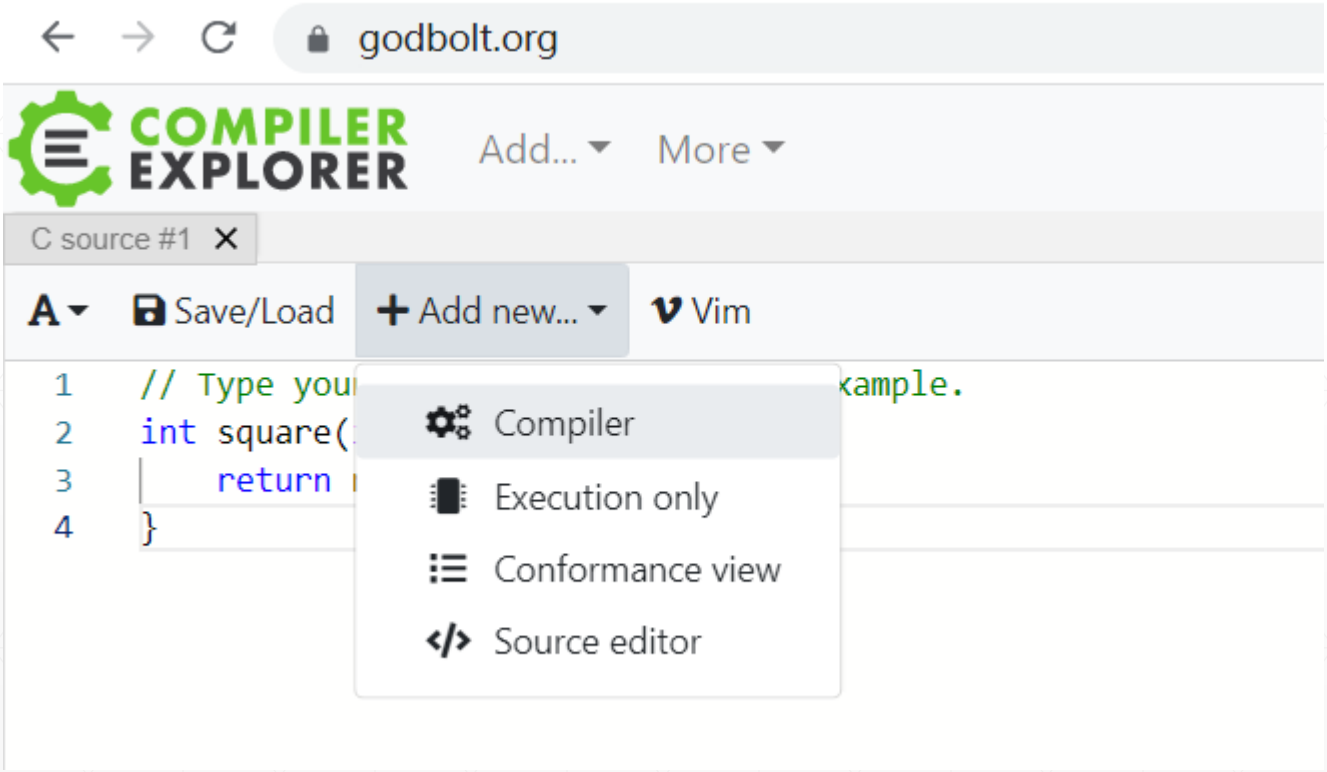
addr	code
1	e_r.code
…	
n	测e_r .code
n+1	j_F, , , m
n+2	S_1.code
…	
m	

语句翻译设计要点

- 语句翻译设计要点
  - 确定语句的目标结构
  - 确定中间代码
  - 根据目标结构和语义规则，构造语义子程序（转换翻译程序）
  - 涉及的实现技术

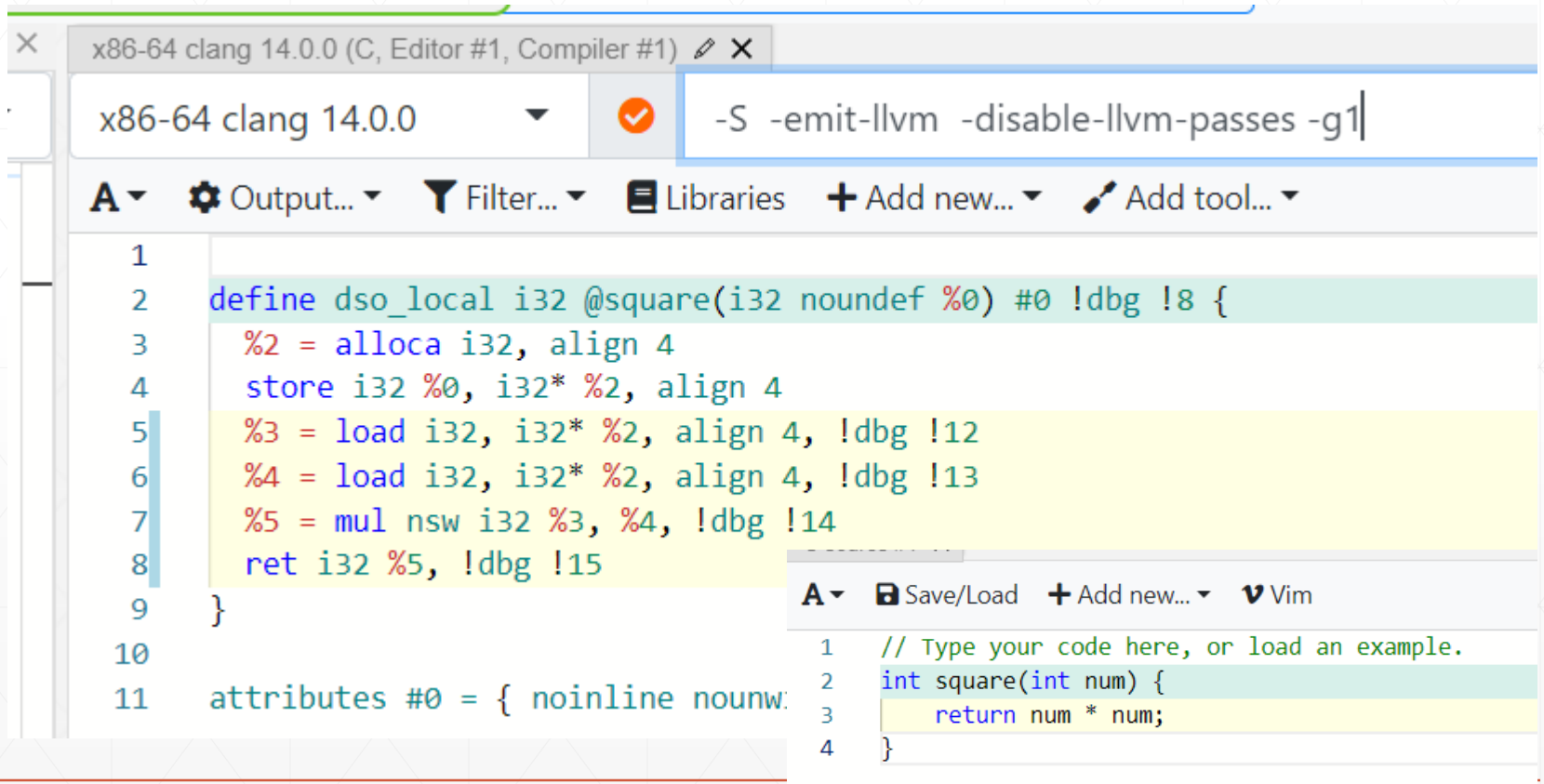
语句翻译

- clang观察中间代码输出



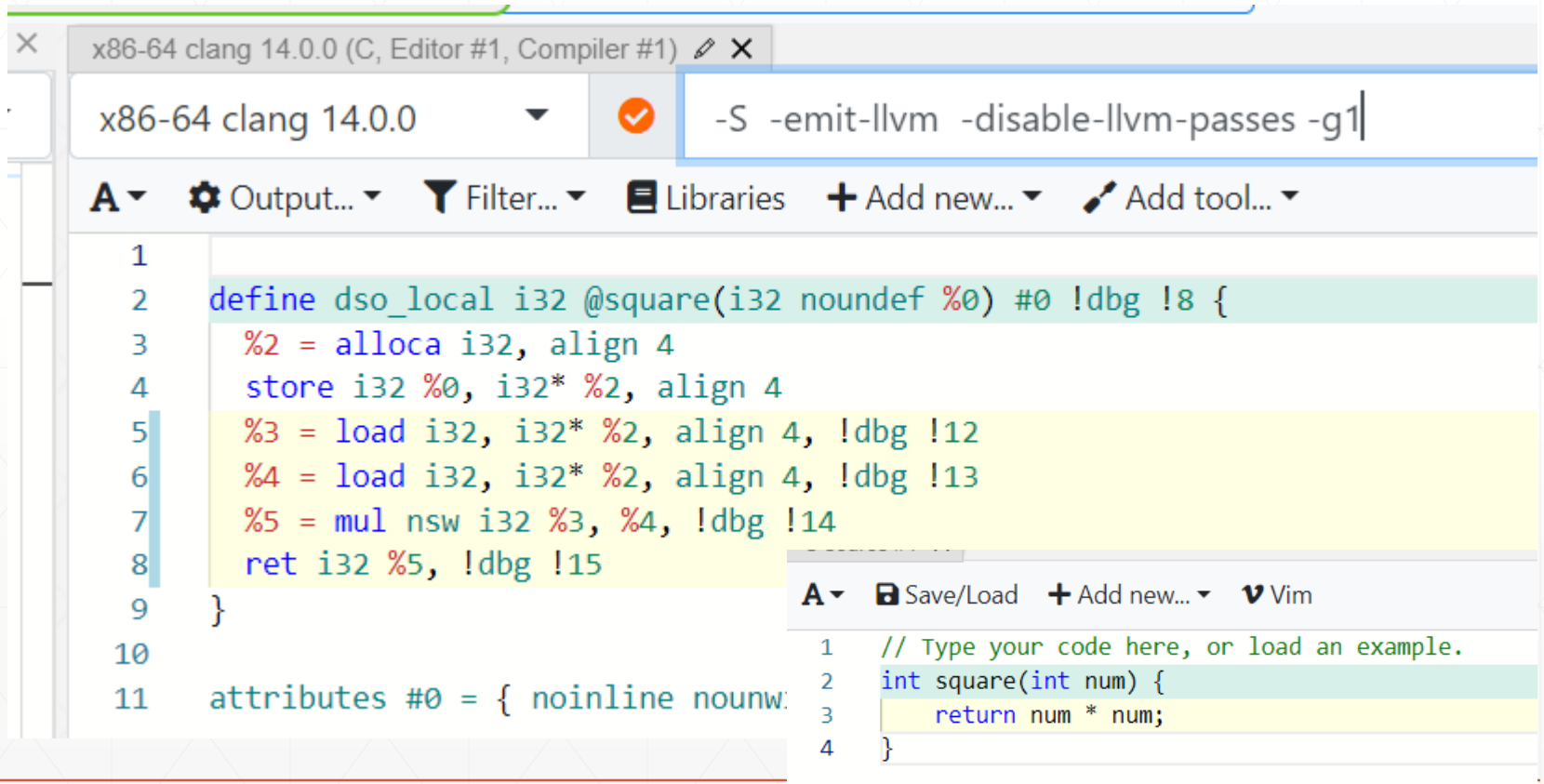
语句翻译

- clang观察中间代码输出



语句翻译

- clang观察中间代码输出



语句翻译

- gcc观察中间代码输出
  - 命令行： gcc -fdump-tree-gimple test.c
  - 输出： test.c.004t.gimple

```
int main(){

        int a;
        int b;

        return 0;
}
```

```
main ()
{
  int D.1760;

  {
    int a;
    int b;

    D.1760 = 0;
    return D.1760;
  }
  D.1760 = 0;
  return D.1760;
}
```

说明类语句

- 语言中定义性信息，一般不产生目标代码，其作用是辅助完成编译。
- 相关说明的属性信息填入符号表，提供语义检查和存储分配的依据。

例如： 变量说明类型说明，对象说明，标号说明 ……

简单说明类语句


int
float
i , j ;
a ,b ;
类型
变量表
namelist
name
kind
type
addr
i
V
I
0
j
V
I
4
a
V
R
8
b
V
R
16
…
…

简单说明类语句

- 翻译方案和语义子程序
  - D.AT: 设为非终结符D的语义变量，它记录说明语句所规定的量的某种性质。
  - fill(P, A): 函数。完成把性质A填入P所指的符号表入口的相应数据项中。
  - ENTRY(i): 函数。给出i所代表的量在符号表中的入口。

S → D ;

D→int id { fill(ENTRY(id)，int)；D.AT=int }

D→float id { fill(ENTRY(id)，float)；D.AT=float }

D→D, id { fill(ENTRY(id)，D1 .AT) D.AT=D1 .AT }

复合类型说明语句

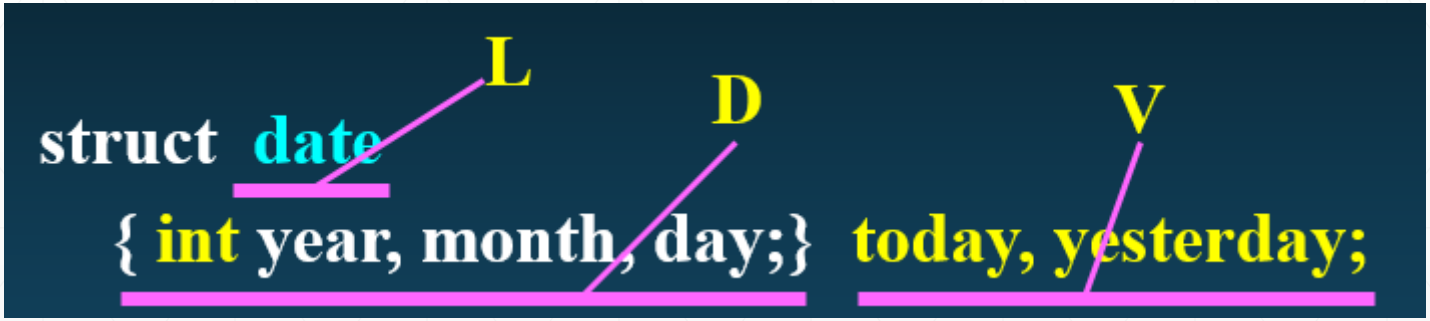
T → struct L{D}[V];

L → id | ε

D → D;F | F

F → type V;

V → V, id | id



复合类型说明语句

```
struct k1 ⇨ L
{
    int  a;
    float  b;
    int c[10];
    char  d;
}
```

V空

{D}

namelist

name	kind	…	addr
k1	struct		
…			

结构成员分表

name	kind	type	LEN	OFFSET
a	V	I	4	0
b	V	R	8	4
c	array	I	40	12
d	V	C	1	52
…				

说明类语句

- LLVM

```
1
2  define dso_local i32 @main() #0 !dbg !14 {
3    %1 = alloca i32, align 4
4    ret i32 0, !dbg !18
5  }
6
```

```
1  int main(){
2      int ret;
3  }
```

- GCC: GIMPLE

```
1  int main(){
2      int ret;
3  }
```

```
main ()
{
  int D.1759;

  {
    int ret;


  }
  D.1759 = 0;
  return D.1759;
}
```

常量语句

Type	Declaration	pointer value change (*ptr = 100)	pointing value change (ptr = &a)
Pointer to Variable	int * ptr	yes	yes
Pointer to Constant	•**const** int * ptr •int **const** * ptr	no	yes
Constant Pointer to Variable	int * **const** ptr	yes	no
Constant Pointer to Constant	**const** int * **const** ptr	no	no

常量语句

```c
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    /* ptr is pointer to constant */
    const int *ptr = &i;

    printf("ptr: %d\n", *ptr);
    /* error: object pointed cannot be modified
    using the pointer ptr */
    *ptr = 100;

    ptr = &j;           /* valid */
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

常量语句

```c
int main(void)
{
    /* i is stored in read only area*/
    int const i = 10;
    int j = 20;

    /* pointer to integer constant. Here i
    is of type "const int", and &i is of
    type "const int *".  And p is of type
    "const int", types are matching no issue */
    int const *ptr = &i;

    printf("ptr: %d\n", *ptr);

    /* error */
    *ptr = 100;

    /* valid. We call it up qualification. In
    C/C++, the type of "int *" is allowed to up
    qualify to the type "const int *". The type of
    &j is "int *" and is implicitly up qualified by
    the compiler to "const int *" */

    ptr = &j;
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

常量语句

```c
int main(void)
{
    int i = 10;
    int const j = 20;

    /* ptr is pointing an integer object */
    int *ptr = &i;

    printf("*ptr: %d\n", *ptr);

    /* The below assignment is invalid in C++, results in error
       In C, the compiler *may* throw a warning, but casting is
       implicitly allowed */
    ptr = &j;

    /* In C++, it is called 'down qualification'. The type of expression
       &j is "const int *" and the type of ptr is "int *". The
       assignment "ptr = &j" causes to implicitly remove const-ness
       from the expression &j. C++ being more type restrictive, will not
       allow implicit down qualification. However, C++ allows implicit
       up qualification. The reason being, const qualified identifiers
       are bound to be placed in read-only memory (but not always). If
       C++ allows above kind of assignment (ptr = &j), we can use 'ptr'
       to modify value of j which is in read-only memory. The
       consequences are implementation dependent, the program may fail
       at runtime. So strict type checking helps clean code. */

    printf("*ptr: %d\n", *ptr);

    return 0;
}
```

常量语句

```c
#include <stdio.h>

int main(void)
{
   int i = 10;
   int j = 20;
/* constant pointer to integer */
   int *const ptr = &i;

   printf("ptr: %d\n", *ptr);

   *ptr = 100;    /* valid */
   printf("ptr: %d\n", *ptr);

   ptr = &j;        /* error */
   return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
/* constant pointer to constant integer */
    const int *const ptr = &i;

    printf("ptr: %d\n", *ptr);

    ptr = &j;      /* error */
    *ptr = 100;    /* error */

    return 0;
}
```

常量语句

CONST_DEF → CONST <con_list>;

<con_list> → <con_list>;CD

<con_list> → CD

CD→id=num

```
{ num.ord=look_con_table(num.lexval);
      id.ord=num.ord;
      id.type= num.type;
      id.kind=constant;
      add(id.entry;id.ord; id.type; id.kind) }
```

常量语句

- LLVM

```
1
2   float area(float r){
3       const float pi = 3.14;
4
5       return pi * r * r;
6
7   }
```

```
1
2   define dso_local float @area(float noundef %0) #0 !dbg !14 {
3     %2 = alloca float, align 4
4     %3 = alloca float, align 4
5     store float %0, ptr %2, align 4
6     store float 0x40091EB860000000, ptr %3, align 4, !dbg !18
7     %4 = load float, ptr %2, align 4, !dbg !19
8     %5 = fmul float 0x40091EB860000000, %4, !dbg !20
9     %6 = load float, ptr %2, align 4, !dbg !21
10    %7 = fmul float %5, %6, !dbg !22
11    ret float %7, !dbg !23
12  }
```

常量语句

- GCC: GIMPLE

```
1
2  float area(float r){
3      const float pi = 3.14;
4
5      return pi * r * r;
6
7  }
```

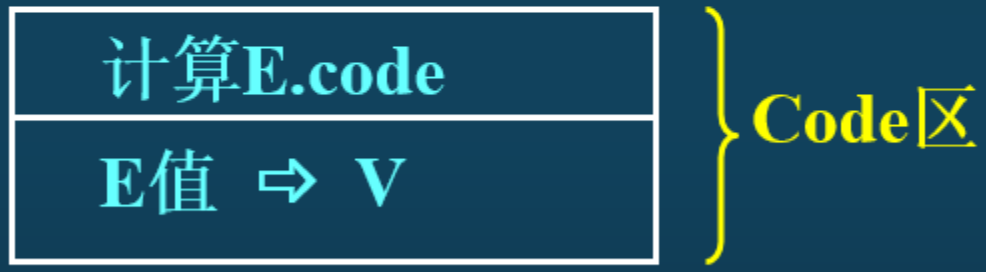
```
area (float r)
{
  float D.1760;
  float D.1761;
  const float pi;

  pi = 3.14000010490417480468 75e+0;
  D.1761 = pi * r;
  D.1760 = D.1761 * r;
  return D.1760;
}
```

赋值语句翻译

- 赋值语句形式定义

$$A \rightarrow V = E$$

- 赋值语句目标结构



** 赋值语句的处理集中在表达式的处理上

赋值语句翻译

- 赋值语句翻译语义子程序

A →i=E

{ GEN(=，E.PLACE，_，ENTRY(i) }

其中:

GEN: 函数。

把四元式(OP，ARG1，ARG2，RESULT)填入四元式表。

E. PLACE：表示存放E值的变量名在符号表的入口地址。

函数ENTRY(i)同前。

表达式翻译

- 表达式形式定义

$$E \rightarrow E_1 \ op \ E_2 \mid op \ E_1 \mid id$$

其中：

OP：为算术运算符；

E_1 , E_2, id：运算对象。

- 表达式语义处理

(1) 表达式处理(产生表达式的中间代码)；

(2) “=”的处理：“=”左右部类型相容性检查和转换；

表达式翻译

- 表达式翻译的语义子程序

(1) $E \rightarrow E_1$ OP E_2

{ E. PLACE=NEWTEMP;

GEN(OP, E_1. PLACE, E_2. PLACE, E.PLACE)}

(2) $E \rightarrow$ OP E_1

{ E. PLACE=NEWTEMP;

GEN(OP, E1. PLACE, _, E.PLACE)}

(3) $E \rightarrow$ id

{ E. PLACE=ENTRY(i) }

赋值和表达式

- LLVM

```
1
2   float area(float r){
3       float ret;
4       const float pi = 3.14;
5
6       ret = pi * r * r;
7       return ret;
8
9   }
```

```
1
2   define dso_local float @area(float noundef %0) #0 !dbg !14 {
3     %2 = alloca float, align 4
4     %3 = alloca float, align 4
5     %4 = alloca float, align 4
6     store float %0, ptr %2, align 4
7     store float 0x40091EB860000000, ptr %4, align 4, !dbg !18
8     %5 = load float, ptr %2, align 4, !dbg !19
9     %6 = fmul float 0x40091EB860000000, %5, !dbg !20
10    %7 = load float, ptr %2, align 4, !dbg !21
11    %8 = fmul float %6, %7, !dbg !22
12    store float %8, ptr %3, align 4, !dbg !23
13    %9 = load float, ptr %3, align 4, !dbg !24
14    ret float %9, !dbg !25
15  }
```

赋值和表达式

- GCC: GIMPLE

```
1
2   float area(float r){
3       float ret;
4       const float pi = 3.14;
5
6       ret = pi * r * r;
7       return ret;
8
9   }
```

```
area (float r)
{
  float D.1761;
  float D.1762;
  float ret;
  const float pi;

  pi = 3.14000010490417480468 75e+0;
  D.1761 = pi * r;
  ret = D.1761 * r;
  D.1762 = ret;
  return D.1762;
}
```

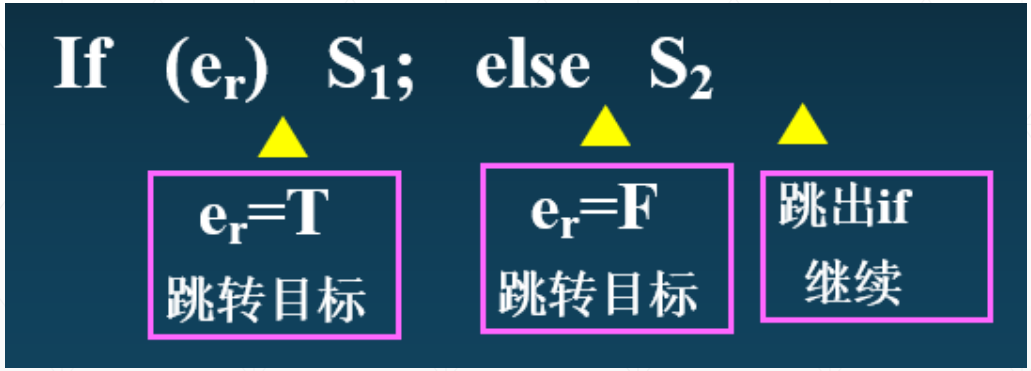
控制流类语句

- 改变程序执行顺序，引起程序执行发生跳转的语句
  - 程序设计语言中出现频繁的语句
  - 为可执行语句，要产生相应的目标代码
  - 控制流程的变换，依靠代码中的跳转指令与对应跳转的语句标号
- 跳转目标与依据

语句标号 {
- **显式：位于源语句之前；（如， L1： goto L2;）**
- **隐式：内含于源语句之中且在源程序中未标识的；**

控制流类语句

- 语句标号处理与拉链-返填技术
  - 控制流类语句处理面对的公共问题和实现技术;
  - 适用于一遍扫描的编译器

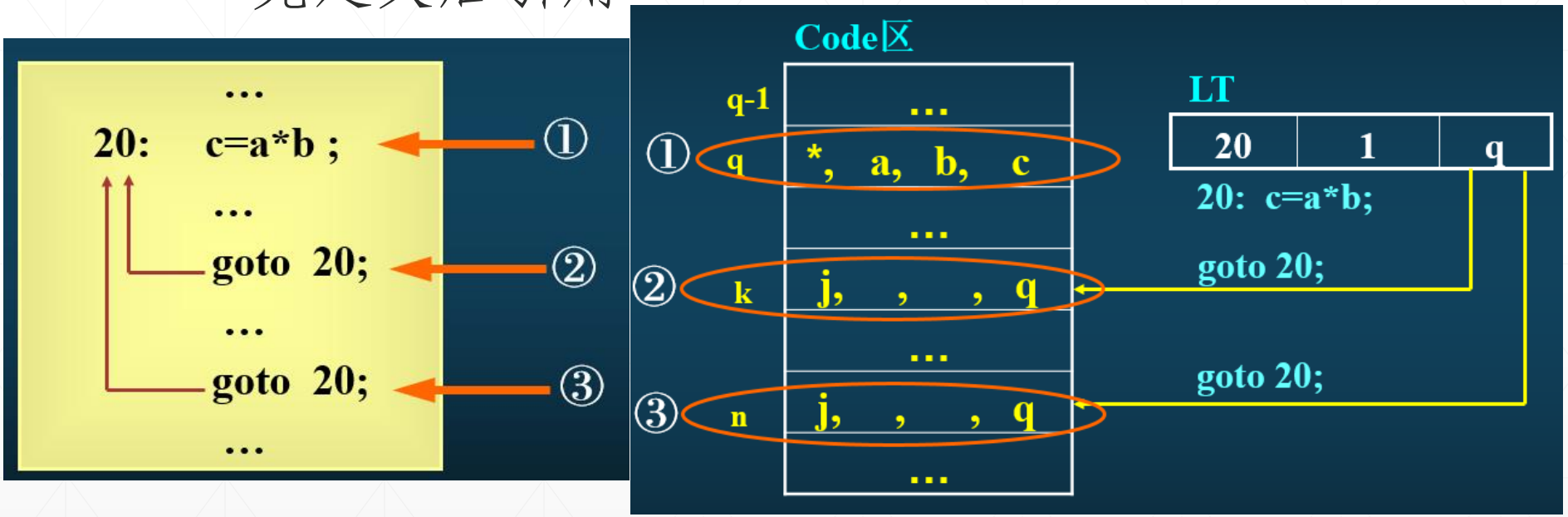


控制流类语句

- 拉链-返填技术

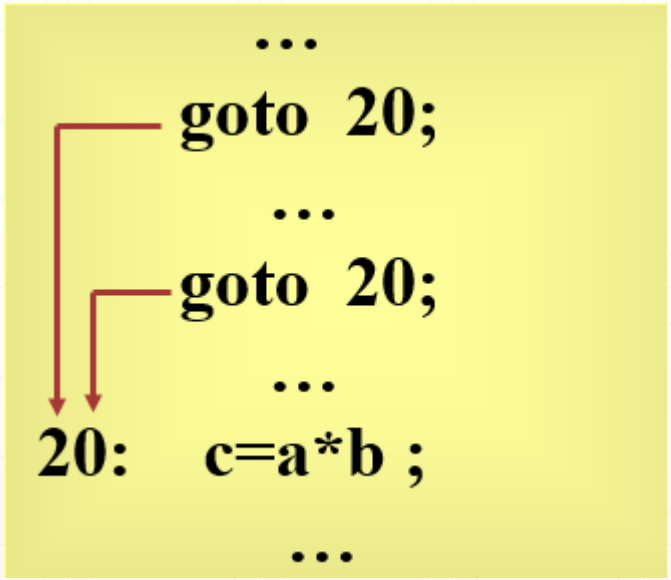
控制流类语句

- 拉链-返填技术
  - 先定义后引用

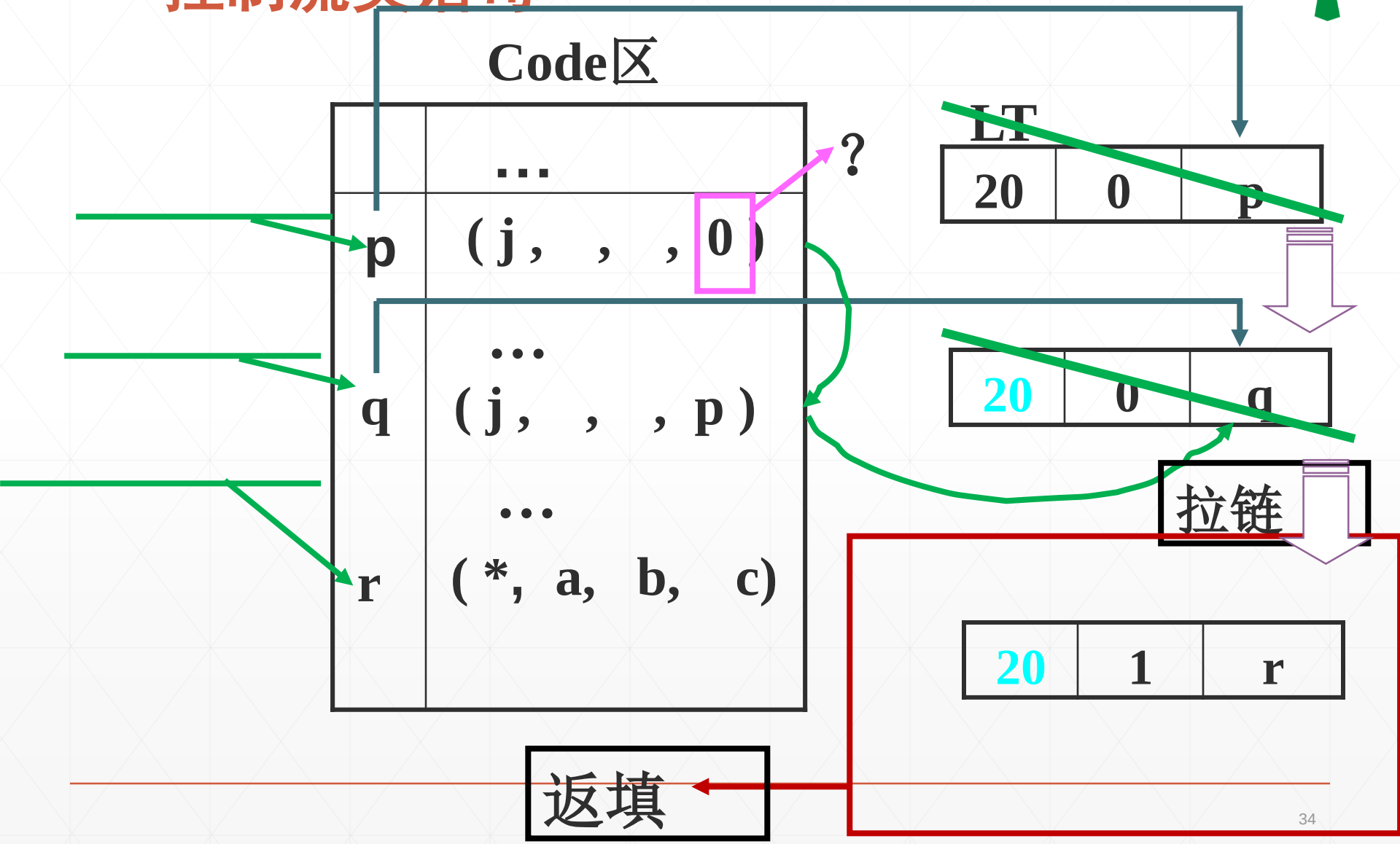


控制流类语句

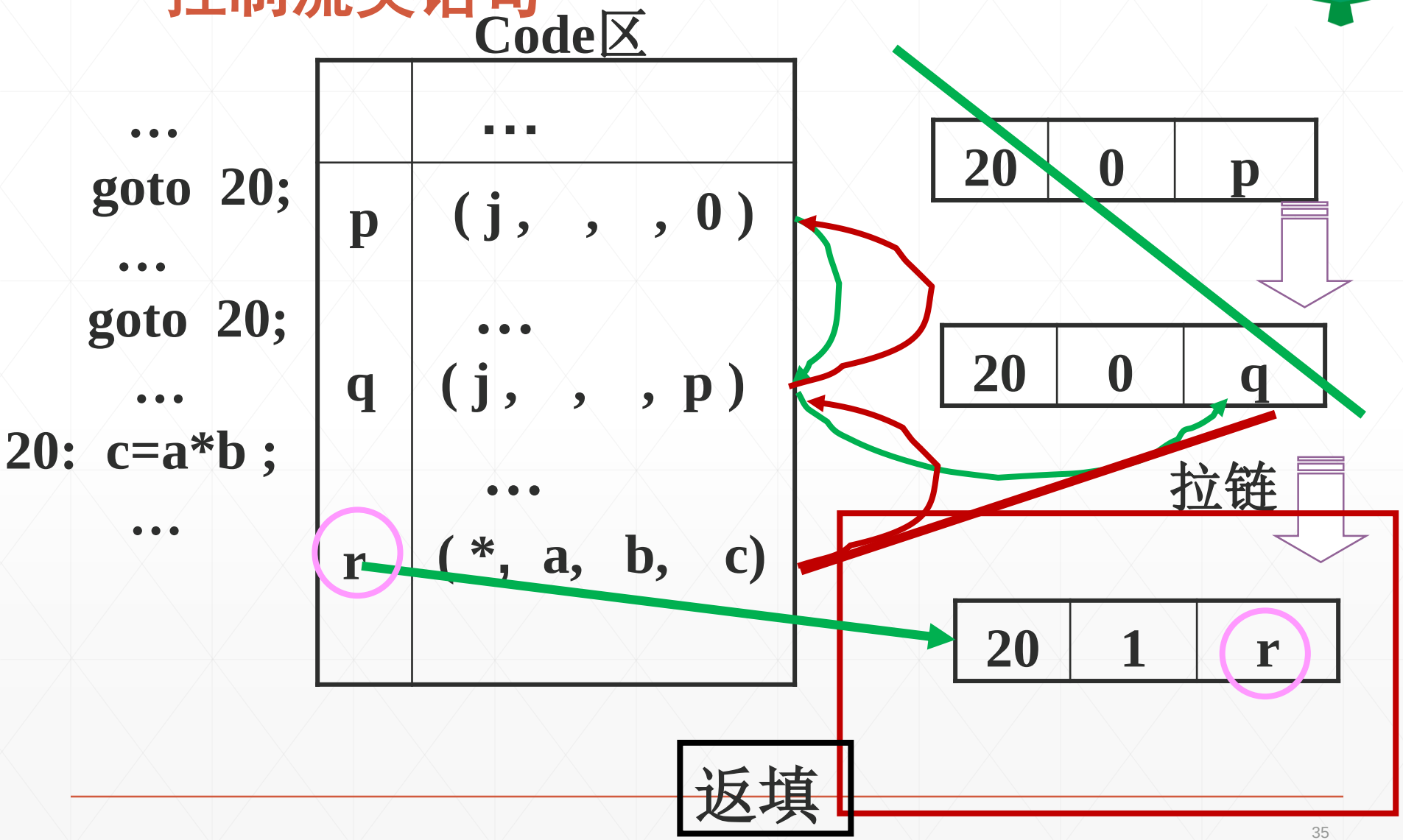
- 拉链-返填技术
  - 先引用后定义



控制流类语句

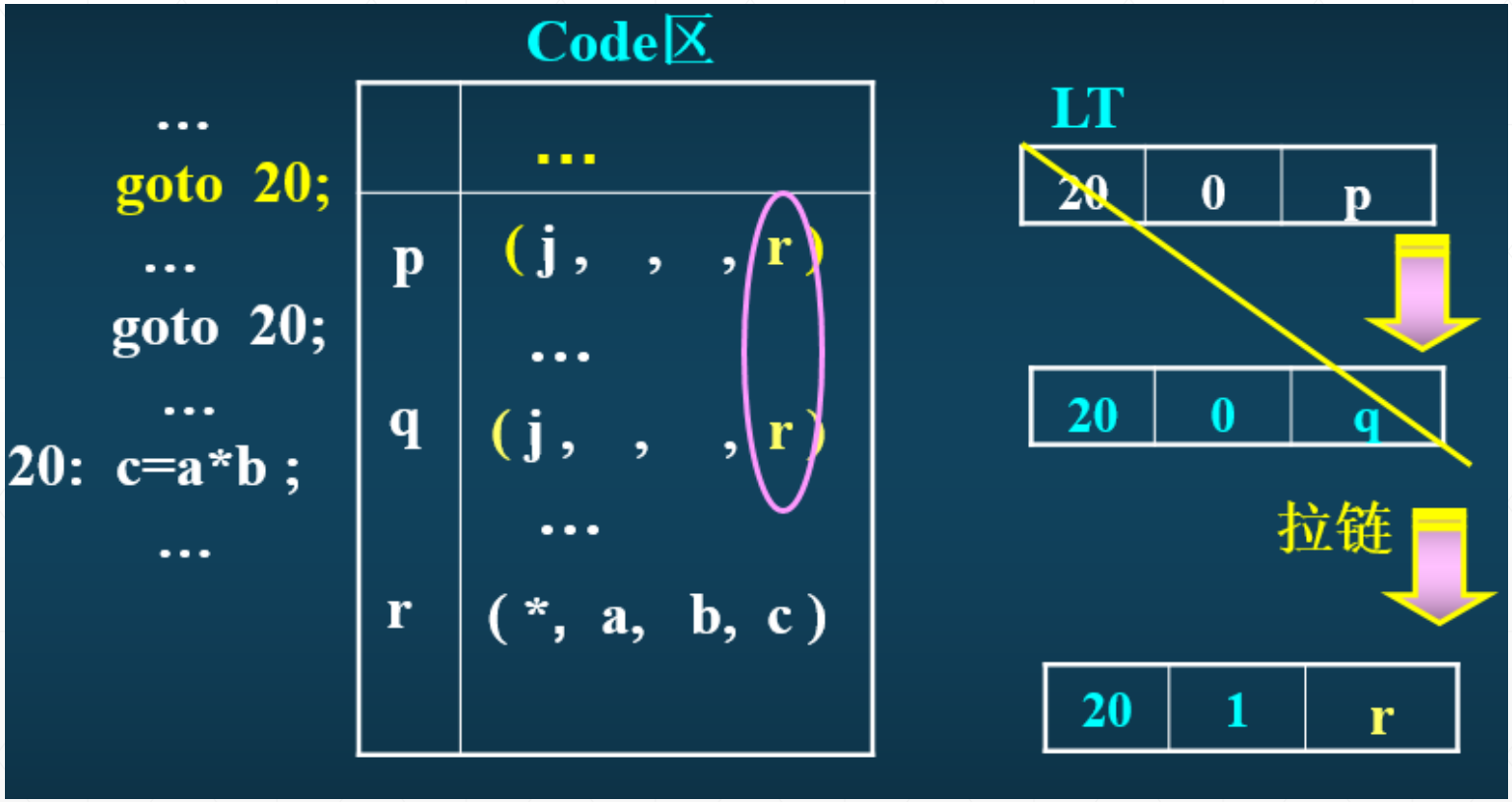


控制流类语句

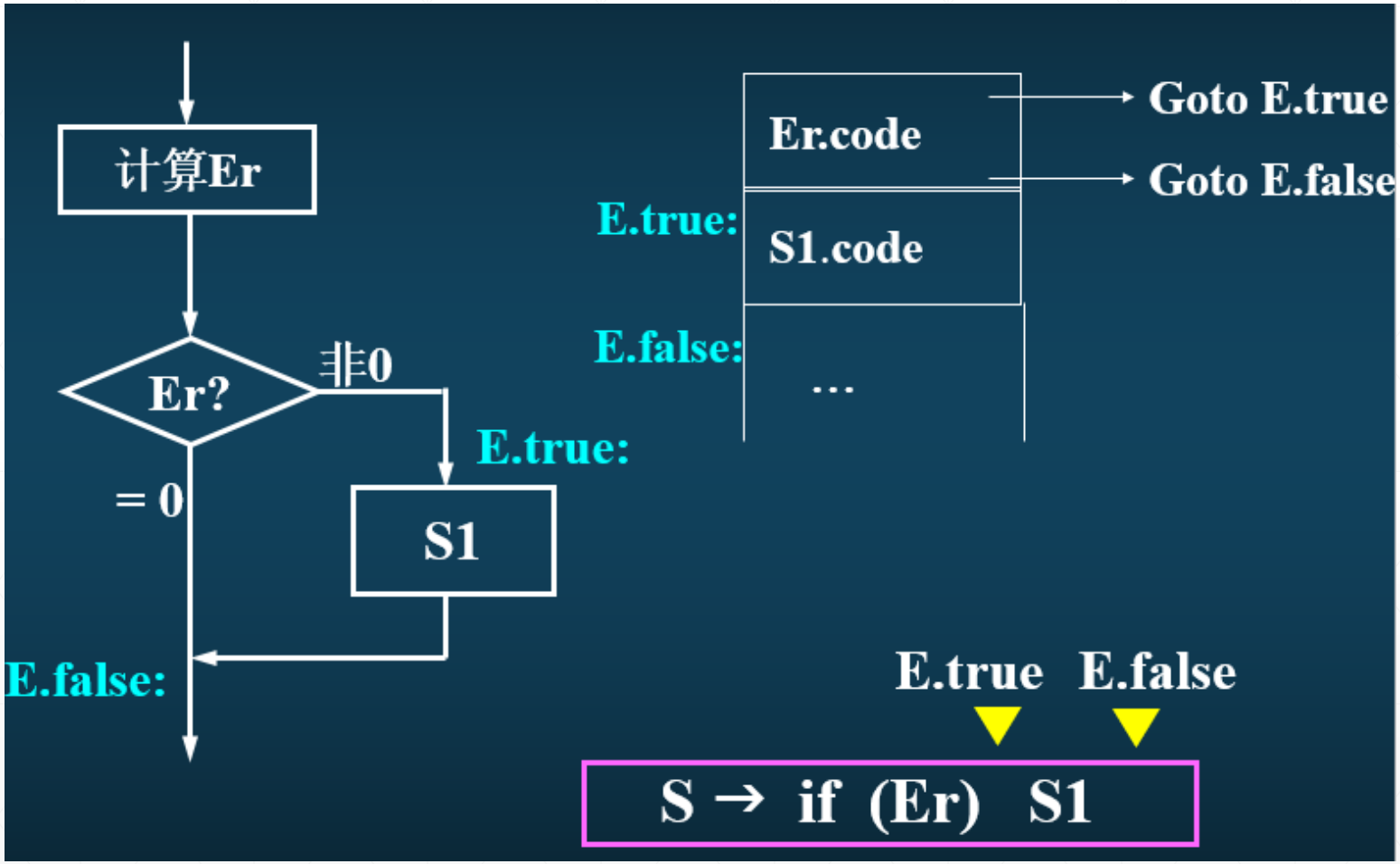


控制流类语句

- 拉链-返填技术



分支语句：if


计算Er
Er?
非0
= 0
E.true:
S1
E.false:
Er.code
Goto E.true
Goto E.false
E.true:
S1.code
E.false:
…
E.true E.false
S → if (Er) S1

分支语句

- LLVM

```c
#define SHAPE_CIRCLE    1
#define SHAPE_SQUQRE    2
#define PI              3.14

float area(int shape, float r){
    float ret;
    if(shape == SHAPE_CIRCLE){
        ret = PI * r * r;
    }else{
        ret = r * r;
    }
    return ret;
}
```

```llvm

define dso_local float @area(i32 noundef %0, float noundef %1)
  %3 = alloca i32, align 4
  %4 = alloca float, align 4
  %5 = alloca float, align 4
  store i32 %0, ptr %3, align 4
  store float %1, ptr %4, align 4
  %6 = load i32, ptr %3, align 4
  %7 = icmp eq i32 %6, 1
  br i1 %7, label %8, label %16

8:                                                ; preds = %2
  %9 = load float, ptr %4, align 4
  %10 = fpext float %9 to double
  %11 = fmul double 3.140000e+00, %10
  %12 = load float, ptr %4, align 4
  %13 = fpext float %12 to double
  %14 = fmul double %11, %13
  %15 = fptrunc double %14 to float
  store float %15, ptr %5, align 4
  br label %20

16:                                               ; preds = %2
  %17 = load float, ptr %4, align 4
  %18 = load float, ptr %4, align 4
  %19 = fmul float %17, %18
  store float %19, ptr %5, align 4
  br label %20

20:                                               ; preds = %16, %8
  %21 = load float, ptr %5, align 4
  ret float %21
}
```

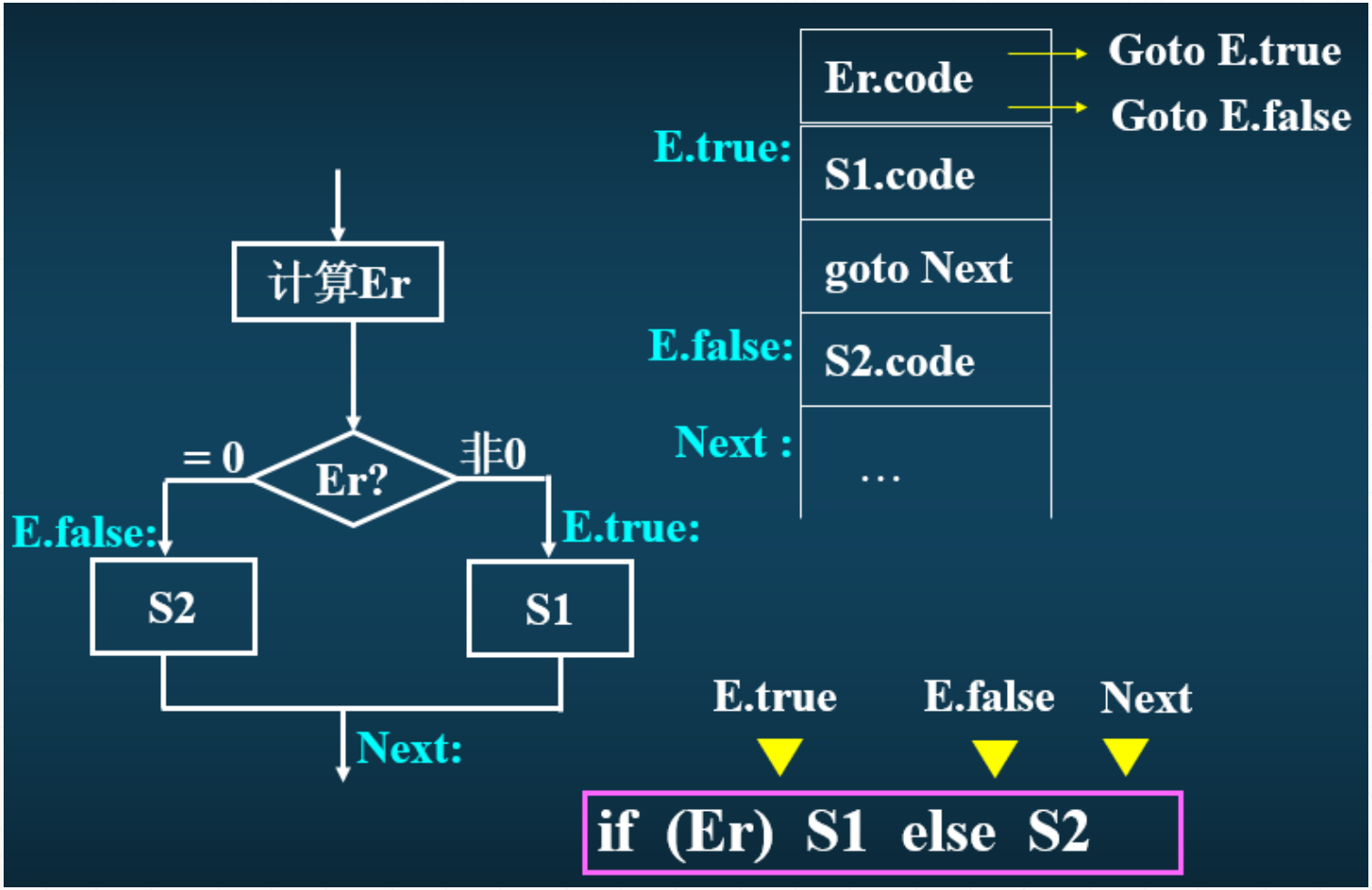
分支语句

- GCC: GIMPLE

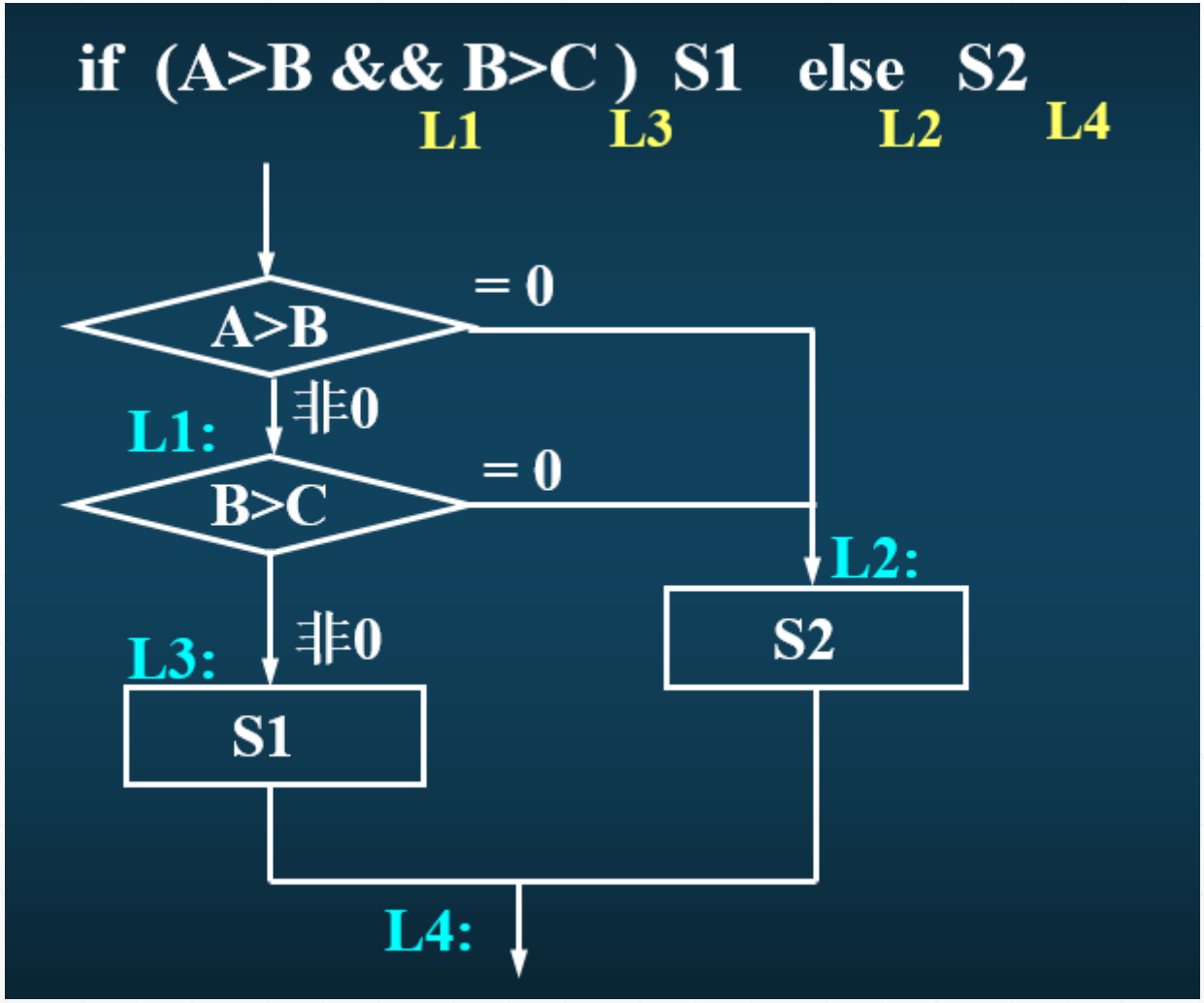
```
1   #define SHAPE_CIRCLE    1
2   #define SHAPE_SQUQRE    2
3   #define PI              3.14
4
5   float area(int shape, float r){
6       float ret;
7       if(shape == SHAPE_CIRCLE){
8           ret = PI * r * r;
9       }else{
10          ret = r * r;
11      }
12      return ret;
13  }
14
```

```
area (int shape, float r)
{
  double D.1763;
  double D.1764;
  double D.1765;
  double D.1766;
  float D.1768;
  float ret;

  if (shape == 1) goto <D.1761>; else goto <D.1762>;
  <D.1761>:
  D.1763 = (double) r;
  D.1764 = D.1763 * 3.14000000000000012434497875801753252744674682617187 5e+0
  D.1765 = (double) r;
  D.1766 = D.1764 * D.1765;
  ret = (float) D.1766;
  goto <D.1767>;
  <D.1762>:
  ret = r * r;
  <D.1767>:
  D.1768 = ret;
  return D.1768;
}
```

分支语句：if-else


计算Er
Er?
= 0
非0
E.false:
E.true:
S2
S1
Next:
Er.code
Goto E.true
Goto E.false
E.true:
S1.code
goto Next
E.false:
S2.code
Next :
…
E.true
E.false
Next
if (Er) S1 else S2

分支语句：if-else

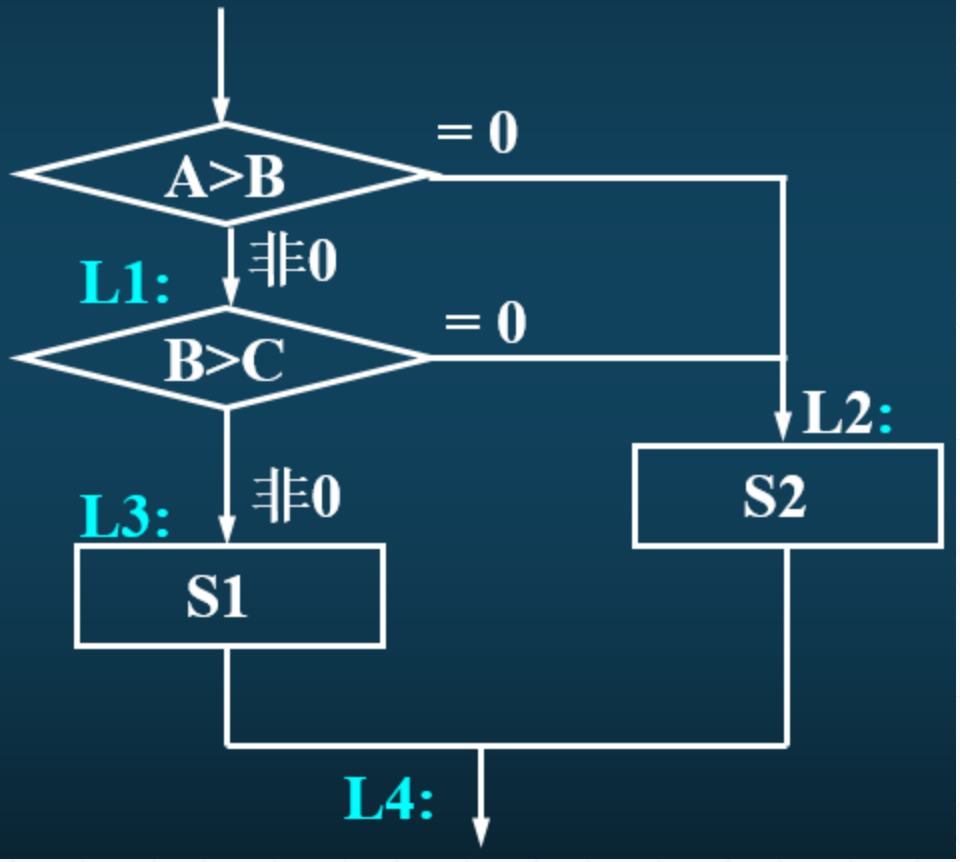


分支语句：if-else

例　if (A>B && B>C) S1 else S2

L1　L3　L2　L4

1	(>, A, B, T_1)
2	(j_F, T_1, , 7)
3	(>, B, C, T_2)
4	(j_F, T_2, , 7)
5	S1.code
6	(j, , , 8)
7	S2.code
8	



多分支语句

- LLVM

```c
#include "stdio.h"

#define SHAPE_CIRCLE    1
#define SHAPE_SQUARE    2
#define PI              3.14

float area(int shape, float r){
    float ret;
    if(shape == SHAPE_CIRCLE){
        ret = PI * r * r;
    }else if(shape == SHAPE_SQUARE){
        ret = r * r;
    }else{
        ret = -1;
    }
    return ret;
}

```

```llvm
define dso_local float @area(i32 noundef %0, float noundef %1) #0 !dbg !14 {
  %3 = alloca i32, align 4
  %4 = alloca float, align 4
  %5 = alloca float, align 4
  store i32 %0, ptr %3, align 4
  store float %1, ptr %4, align 4
  %6 = load i32, ptr %3, align 4, !dbg !18
  %7 = icmp eq i32 %6, 1, !dbg !19
  br i1 %7, label %8, label %16, !dbg !18

8:                                                ; preds = %2
  %9 = load float, ptr %4, align 4, !dbg !20
  %10 = fpext float %9 to double, !dbg !20
  %11 = fmul double 3.140000e+00, %10, !dbg !21
  %12 = load float, ptr %4, align 4, !dbg !22
  %13 = fpext float %12 to double, !dbg !22
  %14 = fmul double %11, %13, !dbg !23
  %15 = fptrunc double %14 to float, !dbg !24
  store float %15, ptr %5, align 4, !dbg !25
  br label %25, !dbg !26

16:                                               ; preds = %2
  %17 = load i32, ptr %3, align 4, !dbg !27
  %18 = icmp eq i32 %17, 2, !dbg !28
  br i1 %18, label %19, label %23, !dbg !27

19:                                               ; preds = %16
  %20 = load float, ptr %4, align 4, !dbg !29
  %21 = load float, ptr %4, align 4, !dbg !30
  %22 = fmul float %20, %21, !dbg !31
  store float %22, ptr %5, align 4, !dbg !32
  br label %24, !dbg !33

23:                                               ; preds = %16
  store float -1.000000e+00, ptr %5, align 4, !dbg !34
  br label %24

24:                                               ; preds = %23, %19
  br label %25

25:                                               ; preds = %24, %8
  %26 = load float, ptr %5, align 4, !dbg !35
  ret float %26, !dbg !36
}
```

多分支语句

- GCC: GIMPLE

```c
#include "stdio.h"

#define SHAPE_CIRCLE    1
#define SHAPE_SQUARE    2
#define PI              3.14

float area(int shape, float r){
    float ret;
    if(shape == SHAPE_CIRCLE){
        ret = PI * r * r;
    }else if(shape == SHAPE_SQUARE){
        ret = r * r;
    }else{
        ret = -1;
    }
    return ret;
}

```

```
area (int shape, float r)
{
  double D.2216;
  double D.2217;
  double D.2218;
  double D.2219;
  float D.2224;
  float ret;

  if (shape == 1) goto <D.2214>; else goto <D.2215>;
  <D.2214>:
  D.2216 = (double) r;
  D.2217 = D.2216 * 3.14000000000000012434497875801753252744670
  D.2218 = (double) r;
  D.2219 = D.2217 * D.2218;
  ret = (float) D.2219;
  goto <D.2220>;
  <D.2215>:
  if (shape == 2) goto <D.2221>; else goto <D.2222>;
  <D.2221>:
  ret = r * r;
  goto <D.2223>;
  <D.2222>:
  ret = -1.0e+0;
  <D.2223>:
  <D.2220>:
  D.2224 = ret;
  return D.2224;
}
```

分支语句：switch

A → switch (e)

{ case c_1 : S_1 break;

case c_2 : S_2 break;

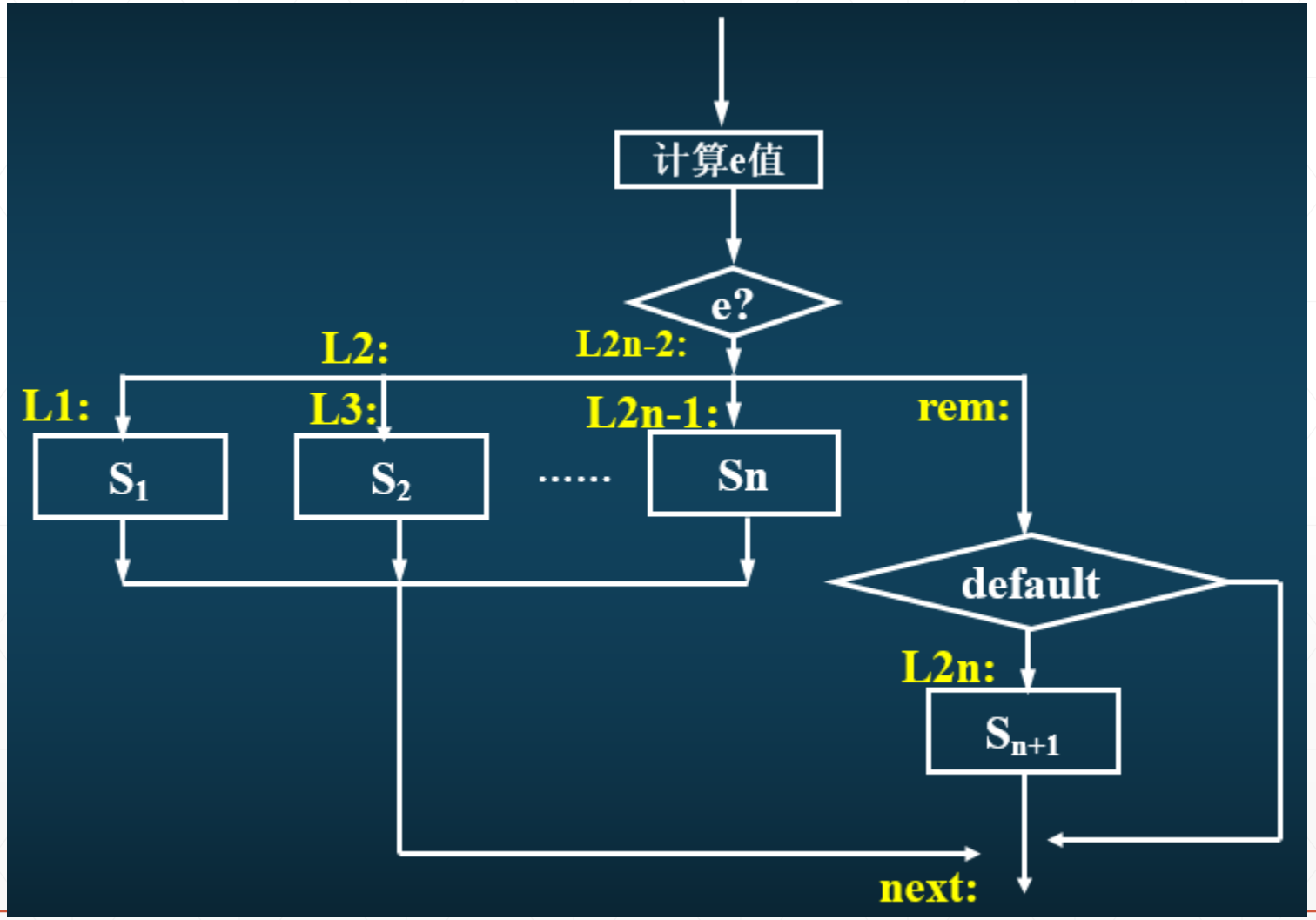
……

case c_n : S_n break;

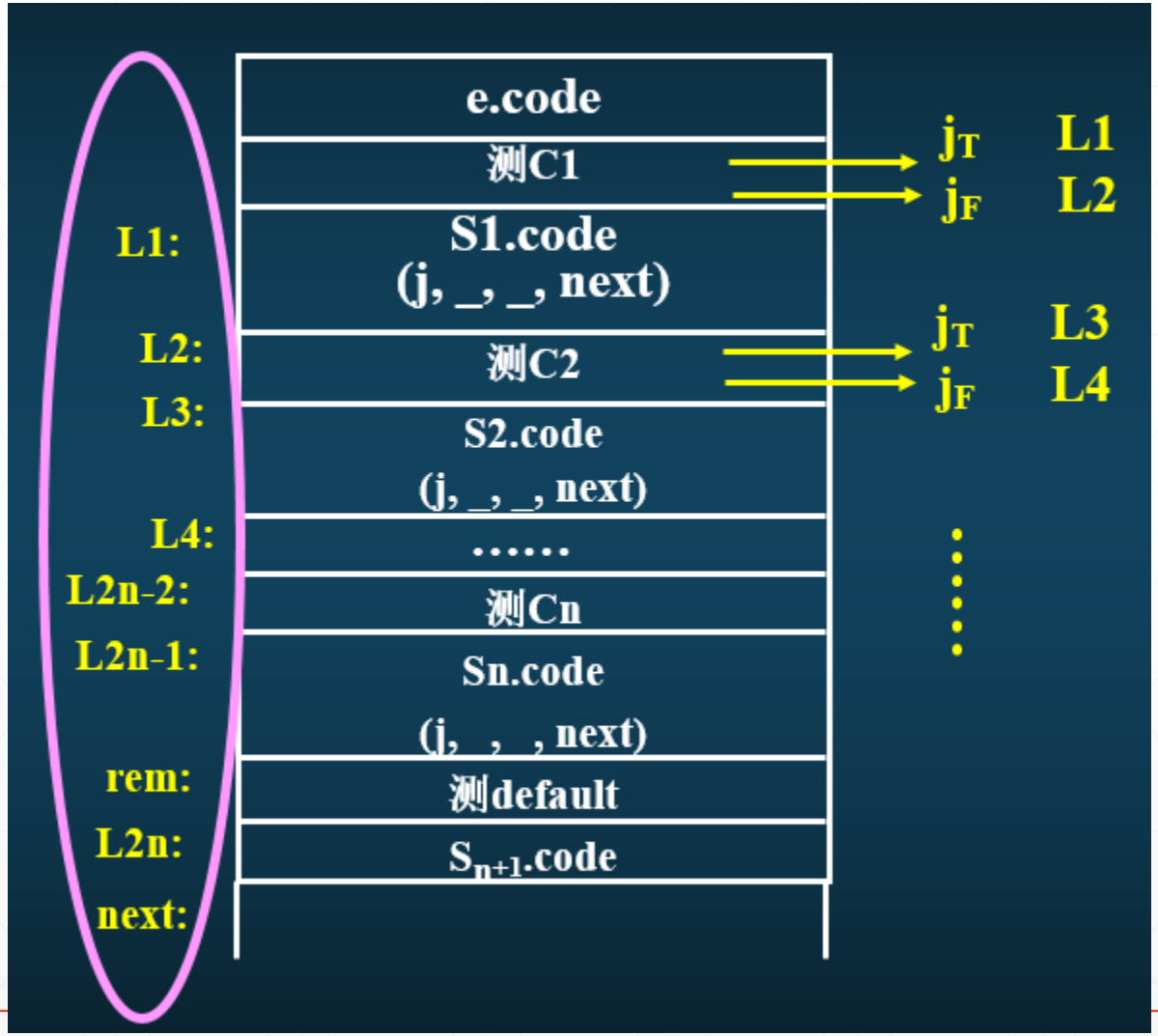
default : S_{n+1}

}

分支语句：switch



分支语句：switch


e.code
测C1
S1.code
(j, _, _, next)
测C2
S2.code
(j, _, _, next)
……
测Cn
Sn.code
(j, , , next)
测default
S_{n+1}.code
L1:
L2:
L3:
L4:
L2n-2:
L2n-1:
rem:
L2n:
next:
j_T L1
j_F L2
j_T L3
j_F L4

分支语句：switch

A → switch (e)

{ case c_1 : S_1 break ;

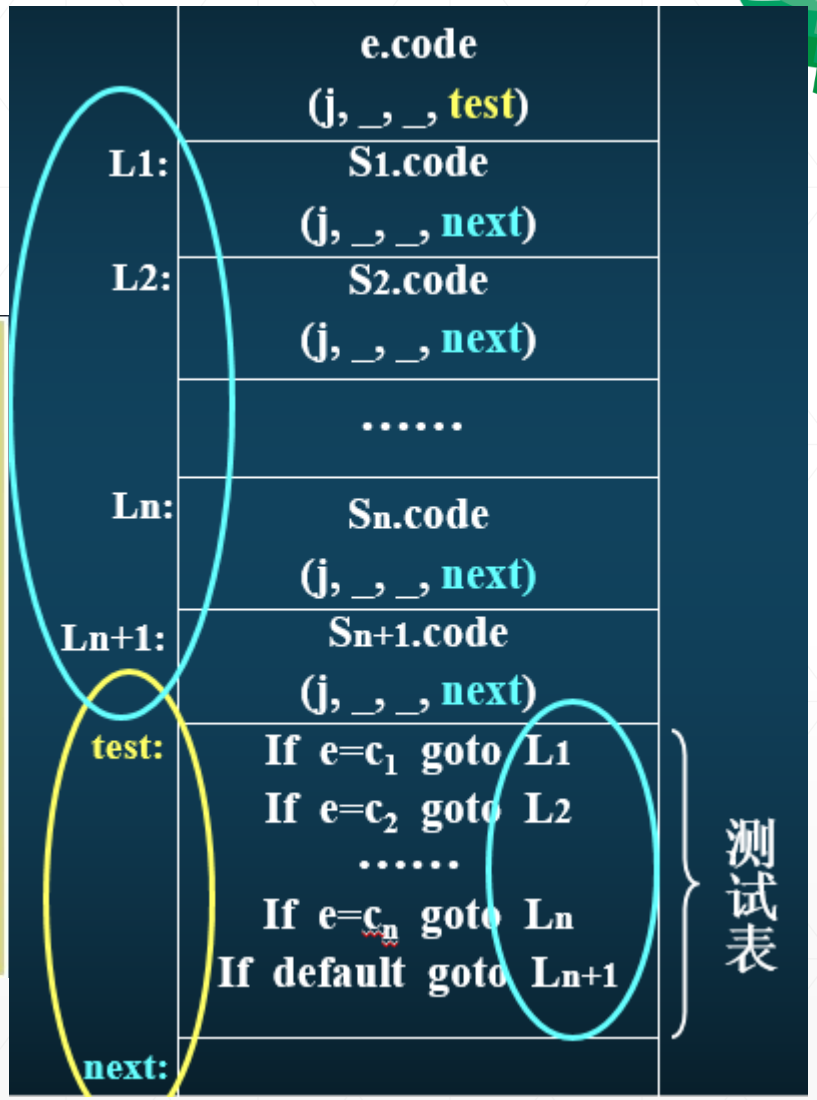
case c_2 : S_2 break ;

……

case c_n : S_n break ;

default : S_{n+1}

}



多分支语句

- LLVM

```c
#include "stdio.h"

#define SHAPE_CIRCLE    1
#define SHAPE_SQUARE    2
#define PI              3.14

float area(int shape, float r){
    float ret;
    switch(shape){
        case SHAPE_CIRCLE:
            ret = PI * r * r;
            break;
        case SHAPE_SQUARE:
            ret = r * r;
            break;
        default:
            ret = -1;
    }
    return ret;
}

```

```llvm

define dso_local float @area(i32 noundef %0, float noundef %1) #0 !dbg !14 {
  %3 = alloca i32, align 4
  %4 = alloca float, align 4
  %5 = alloca float, align 4
  store i32 %0, ptr %3, align 4
  store float %1, ptr %4, align 4
  %6 = load i32, ptr %3, align 4, !dbg !18
  switch i32 %6, label %19 [
    i32 1, label %7
    i32 2, label %15
  ], !dbg !19

7:                                                ; preds = %2
  %8 = load float, ptr %4, align 4, !dbg !20
  %9 = fpext float %8 to double, !dbg !20
  %10 = fmul double 3.140000e+00, %9, !dbg !21
  %11 = load float, ptr %4, align 4, !dbg !22
  %12 = fpext float %11 to double, !dbg !22
  %13 = fmul double %10, %12, !dbg !23
  %14 = fptrunc double %13 to float, !dbg !24
  store float %14, ptr %5, align 4, !dbg !25
  br label %20, !dbg !26

15:                                               ; preds = %2
  %16 = load float, ptr %4, align 4, !dbg !27
  %17 = load float, ptr %4, align 4, !dbg !28
  %18 = fmul float %16, %17, !dbg !29
  store float %18, ptr %5, align 4, !dbg !30
  br label %20, !dbg !31

19:                                               ; preds = %2
  store float -1.000000e+00, ptr %5, align 4, !dbg !32
  br label %20, !dbg !33

20:                                               ; preds = %19, %15, %7
  %21 = load float, ptr %5, align 4, !dbg !34
  ret float %21, !dbg !35
}
```

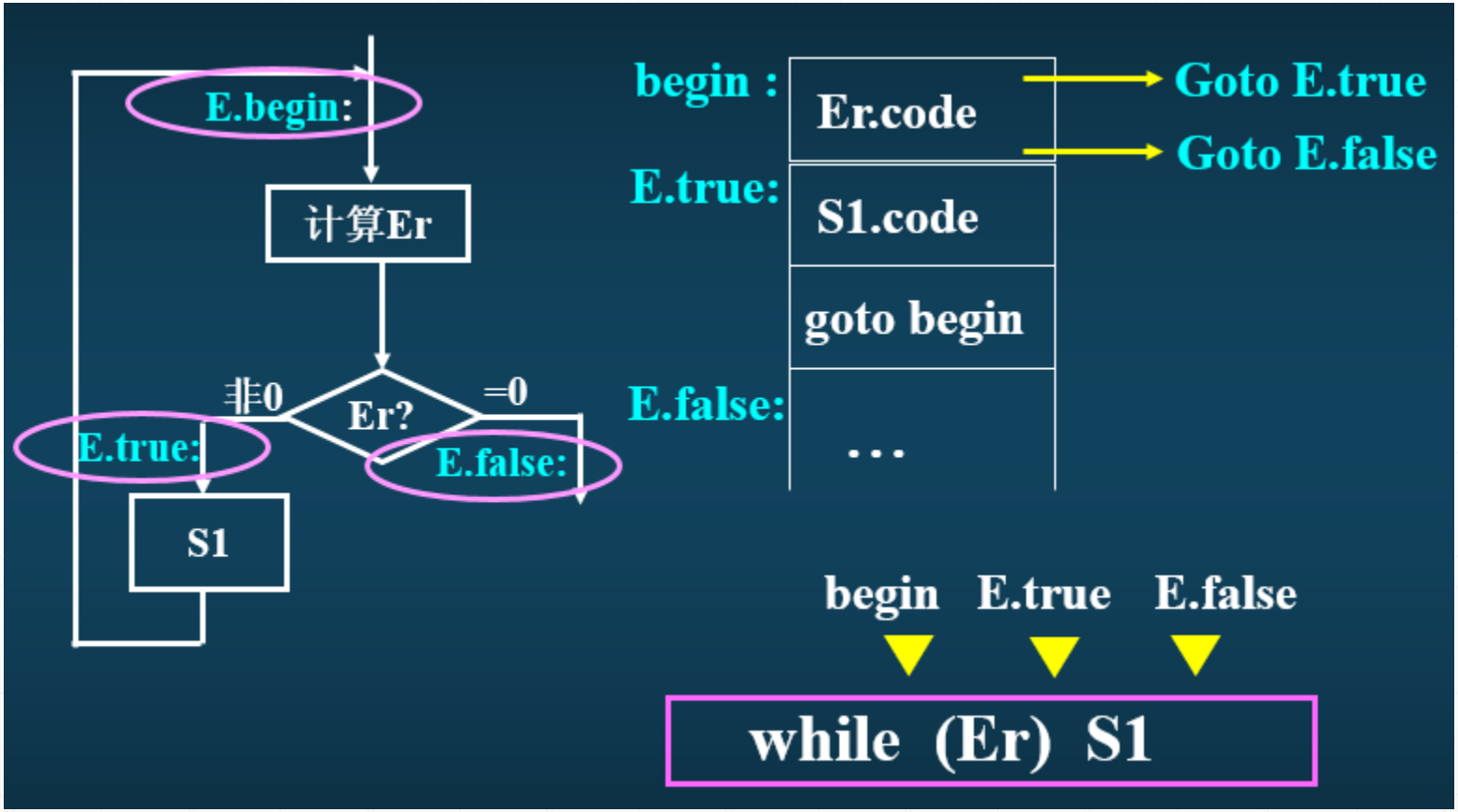
多分支语句

- GCC: GIMPLE

```c
#include "stdio.h"

#define SHAPE_CIRCLE    1
#define SHAPE_SQUARE    2
#define PI              3.14

float area(int shape, float r){
    float ret;
    switch(shape){
        case SHAPE_CIRCLE:
            ret = PI * r * r;
            break;
        case SHAPE_SQUARE:
            ret = r * r;
            break;
        default:
            ret = -1;
    }
    return ret;
}
```

```
area (int shape, float r)
{
  double D.2218;
  double D.2219;
  double D.2220;
  double D.2221;
  float D.2222;
  float ret;

  switch (shape) <default: <D.2216>, case 1: <D.2213>, case 2: <D.2215>>
  <D.2213>:
  D.2218 = (double) r;
  D.2219 = D.2218 * 3.14000000000000012434497875801753252744674682617187 5e+0;
  D.2220 = (double) r;
  D.2221 = D.2219 * D.2220;
  ret = (float) D.2221;
  goto <D.2214>;
  <D.2215>:
  ret = r * r;
  goto <D.2214>;
  <D.2216>:
  ret = -1.0e+0;
  <D.2214>:
  D.2222 = ret;
  return D.2222;
}
```

循环语句：while


E.begin:
计算Er
非0
Er?
=0
E.true:
E.false:
S1
begin :
Er.code
Goto E.true
Goto E.false
E.true:
S1.code
goto begin
E.false:
...
begin E.true E.false
while (Er) S1

while循环语句

- LLVM

```
1   int main(){
2       int i = 10;
3       while(i > 0){
4           i--;
5       }
6       return i;
7   }
```

```
2   define dso_local i32 @main() #0 !dbg !14 {
3     %1 = alloca i32, align 4
4     %2 = alloca i32, align 4
5     store i32 0, ptr %1, align 4
6     store i32 10, ptr %2, align 4, !dbg !18
7     br label %3, !dbg !19
8
9   3:
10    %4 = load i32, ptr %2, align 4, !dbg !20
11    %5 = icmp sgt i32 %4, 0, !dbg !21
12    br i1 %5, label %6, label %9, !dbg !19
13
14  6:
15    %7 = load i32, ptr %2, align 4, !dbg !22
16    %8 = add nsw i32 %7, -1, !dbg !22
17    store i32 %8, ptr %2, align 4, !dbg !22
18    br label %3, !dbg !19, !llvm.loop !23
19
20  9:
21    %10 = load i32, ptr %2, align 4, !dbg !26
22    ret i32 %10, !dbg !27
23  }
```

while循环语句

- GCC: GIMPLE

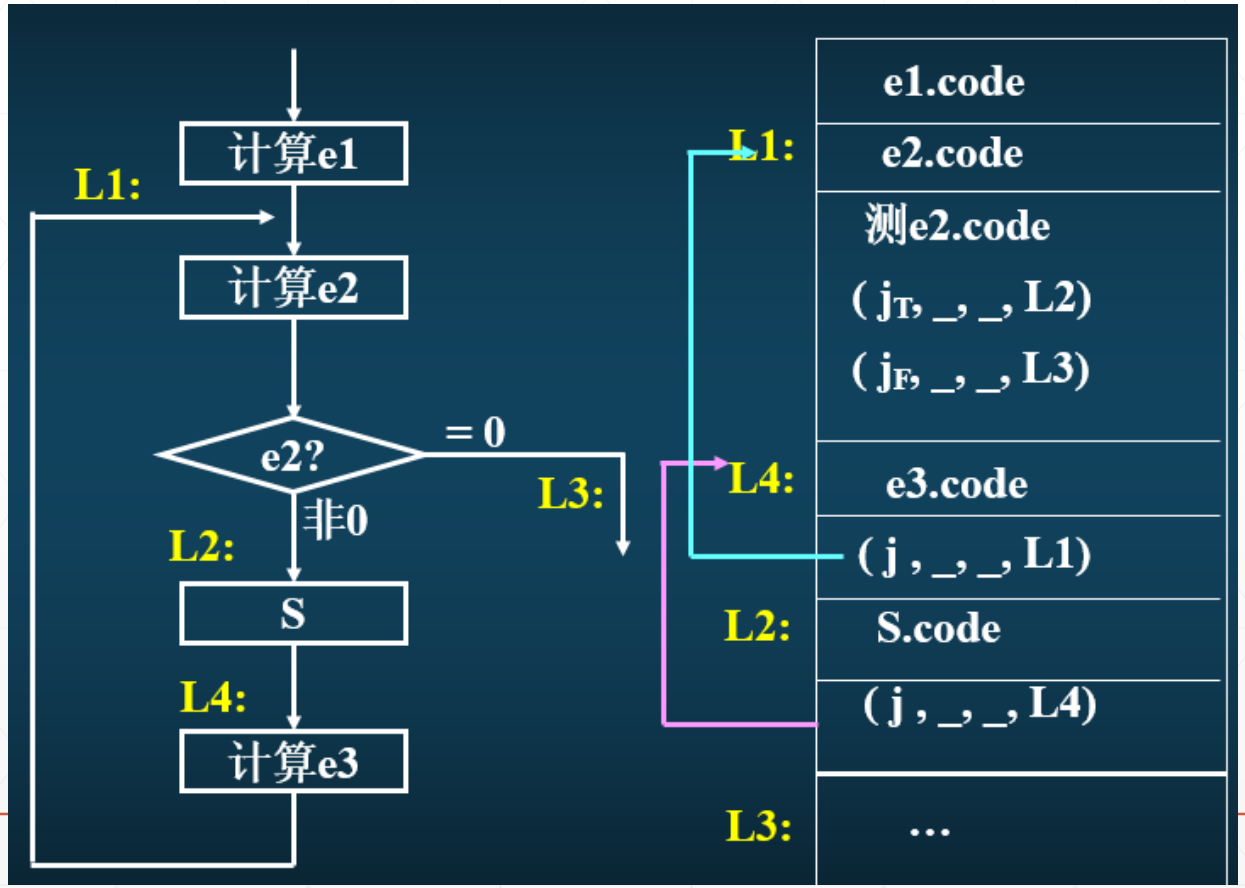
```
1  int main(){
2      int i = 10;
3      while(i > 0){
4          i--;
5      }
6      return i;
7  }
```

```
main ()
{
  int D.1762;

  {
    int i;

    i = 10;
    goto <D.1759>;
    <D.1758>:
    i = i + -1;
    <D.1759>:
    if (i > 0) goto <D.1758>; else goto <D.1760>;
    <D.1760>:
    D.1762 = i;
    return D.1762;
  }
  D.1762 = 0;
  return D.1762;
}
```

循环语句：for

A → for (e1 ; e2 ; e3) S

L1 L4 L2 L3



for循环语句

- LLVM

```c
int main(){
    int i ;

    for(i = 0; i < 10; i++){
        printf("%d\n", i);
    }
}
```

```
define dso_local i32 @main() #0 !dbg !36 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 0, ptr %2, align 4, !dbg !37
  br label %3, !dbg !38

3:                                                ; preds = %9, %0
  %4 = load i32, ptr %2, align 4, !dbg !39
  %5 = icmp slt i32 %4, 10, !dbg !40
  br i1 %5, label %6, label %12, !dbg !41

6:                                                ; preds = %3
  %7 = load i32, ptr %2, align 4, !dbg !42
  %8 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %7), !dbg !43
  br label %9, !dbg !44

9:                                                ; preds = %6
  %10 = load i32, ptr %2, align 4, !dbg !45
  %11 = add nsw i32 %10, 1, !dbg !45
  store i32 %11, ptr %2, align 4, !dbg !45
  br label %3, !dbg !41, !llvm.loop !46

12:                                               ; preds = %3
  %13 = load i32, ptr %1, align 4, !dbg !48
  ret i32 %13, !dbg !48
}
```

for循环语句

- GCC: GIMPLE

```c
int main(){
    int i ;

    for(i = 0; i < 10; i++){
        printf("%d\n", i);
    }
}
```

```
main ()
{
  int D.2230;

  {
    int i;

    i = 0;
    goto <D.2221>;
    <D.2220>:
    printf ("%d\n", i);
    i = i + 1;
    <D.2221>:
    if (i <= 9) goto <D.2220>; else goto <D.2222>;
    <D.2222>:
  }
  D.2230 = 0;
  return D.2230;
}
```

中间代码生成

给出如下C程序段的目标代码结构：

…

for (e1; e2; e3;)

L1 L2

for (t1; t2; t3;)

L3 L4 L5

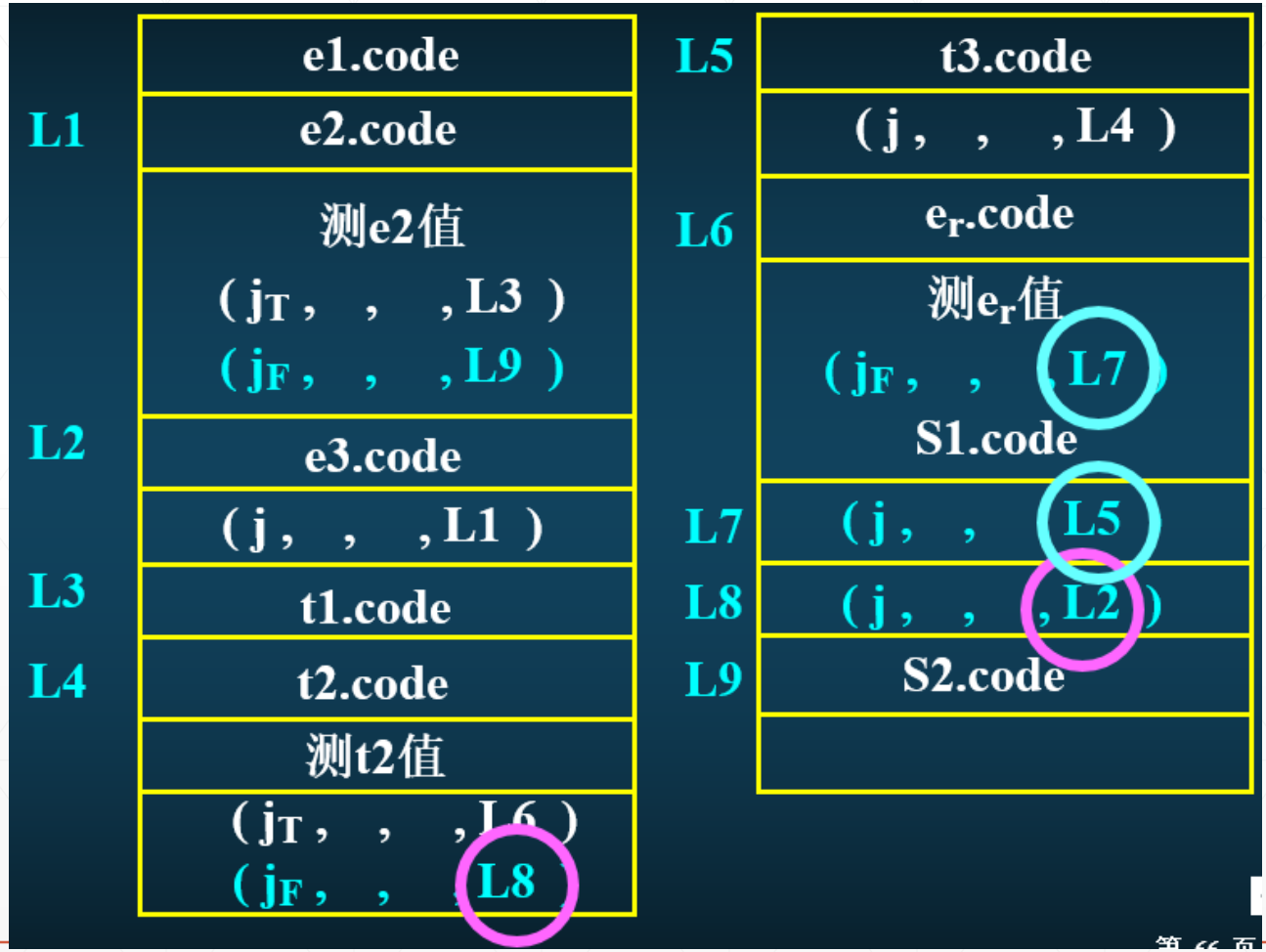
if (e_r) S_1; $L7_{for}$ // S_1是C语句

L6

$L8_{for}$ L9 S_2; // S2是C语句

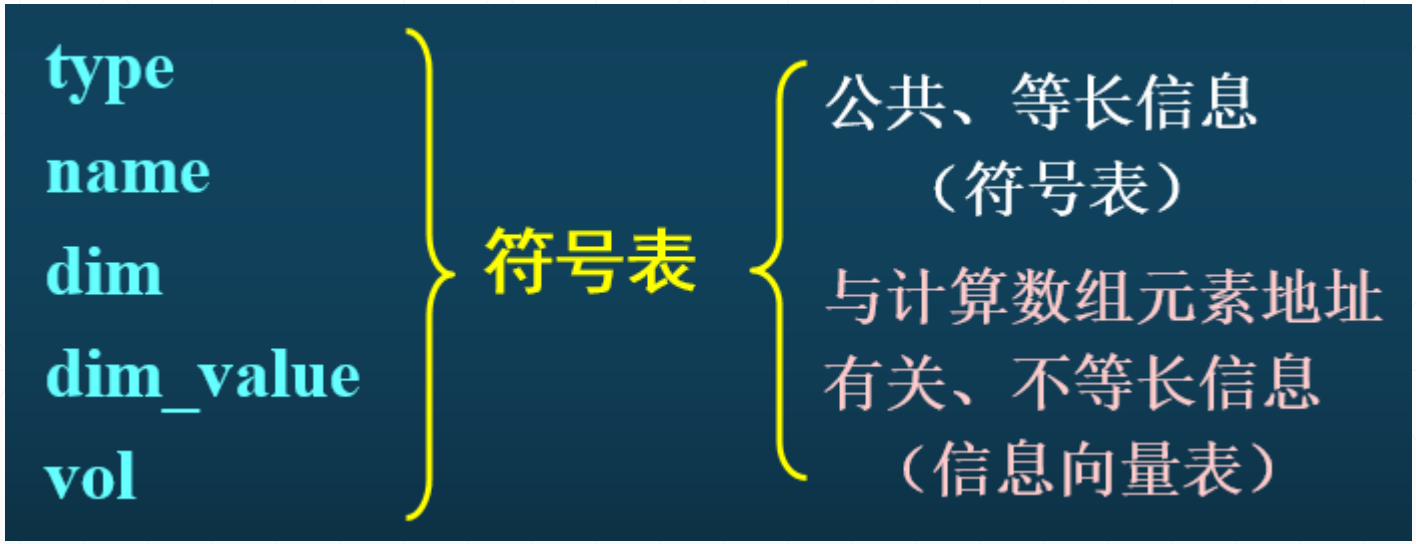
…

中间代码生成


e1.code
L1
e2.code
测e2值
(jT , , , L3)
(jF , , , L9)
L2
e3.code
(j , , , L1)
L3
t1.code
L4
t2.code
测t2值
(jT , , , L6)
(jF , , L8
L5
t3.code
(j , , , L4)
L6
er.code
测er值
(jF , , L7
S1.code
L7
(j , , L5
L8
(j , , , L2)
L9
S2.code

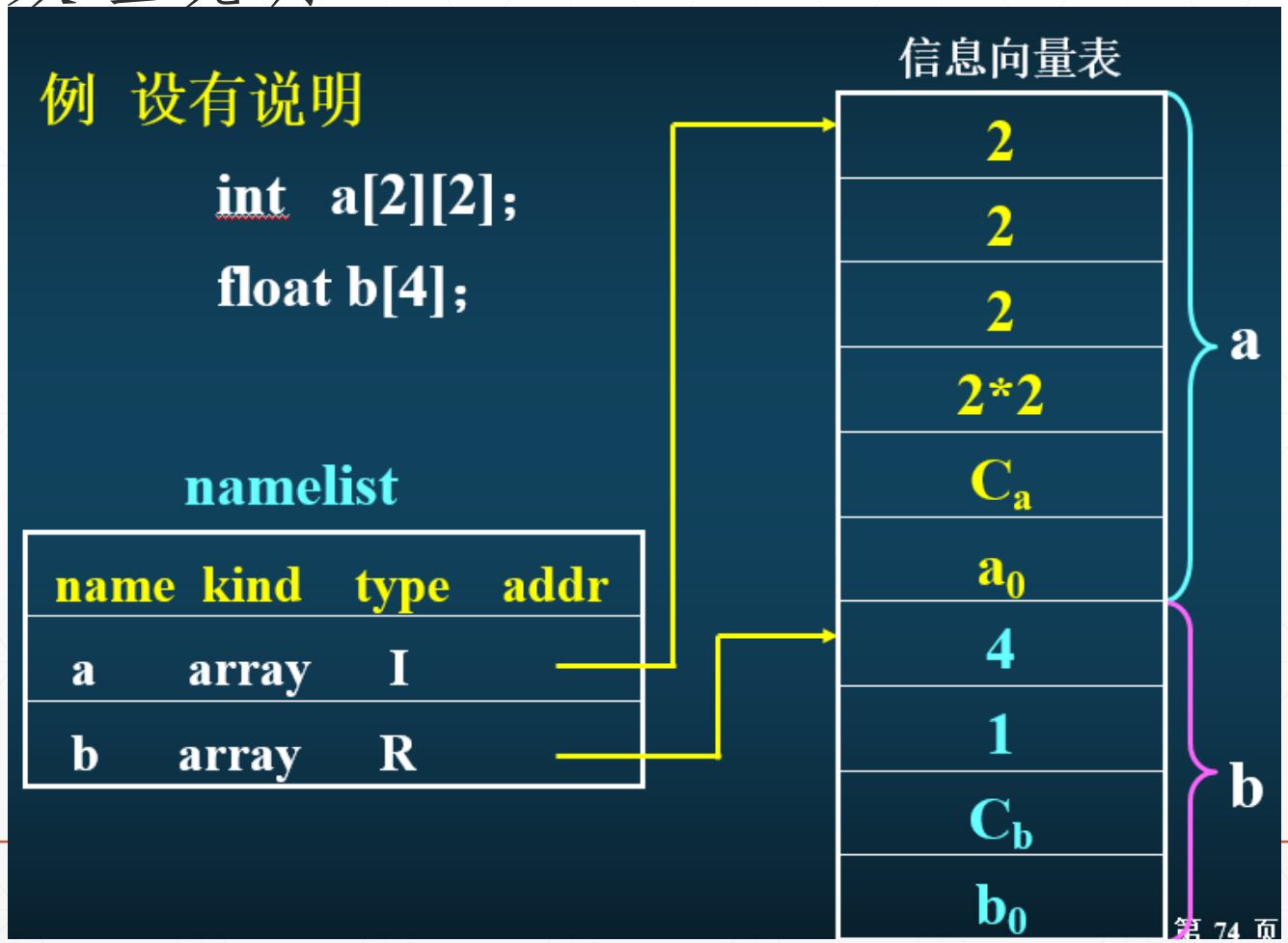
数组的翻译

- 数组说明


type
name
dim
dim_value
vol
符号表
公共、等长信息
（符号表）
与计算数组元素地址
有关、不等长信息
（信息向量表）

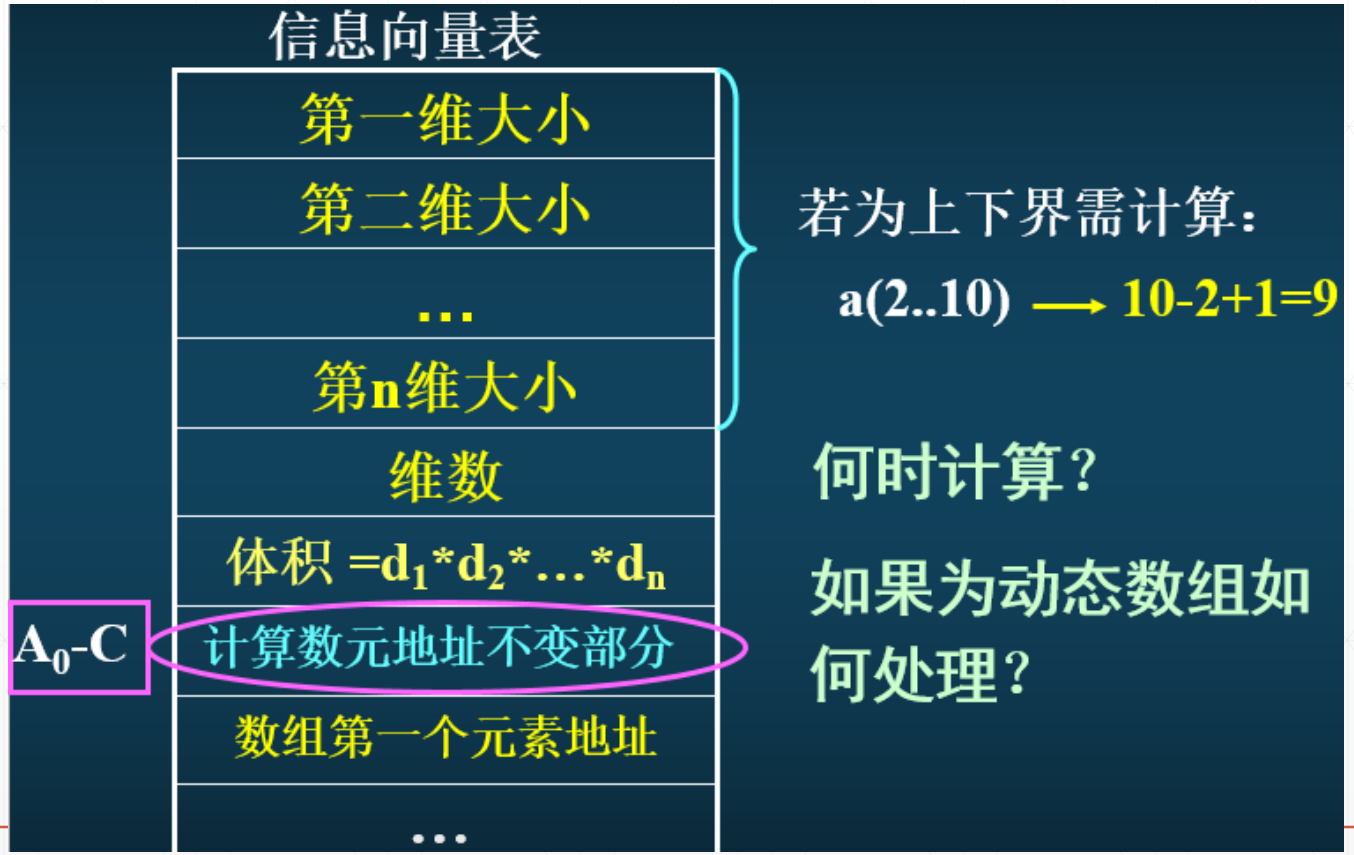
数组的翻译

- 数组说明


例 设有说明
int a[2][2];
float b[4];
namelist
name kind type addr
a array I
b array R
信息向量表
2
2
2
2*2
C_a
a_0
4
1
C_b
b_0
a
b
第 74 页

数组的翻译

- 数组说明


信息向量表
第一维大小
第二维大小
…
第n维大小
维数
体积 =d_1*d_2*…*d_n
A_0-C
计算数元地址不变部分
数组第一个元素地址
…
若为上下界需计算：
a(2..10) ⟶ 10-2+1=9
何时计算？
如果为动态数组如何处理？

数组的翻译

- 数组引用
  - 语义检查： 类型匹配；下标越界检查；
  - 产生代码：数组元素地址计算的中间代码
  - 数组元素地址计算编译时能否完成？

A $[i_1]$ $[i_2]$… $[i_n]$ 数组引用

** 不能整体引用，仅对单一数组元素引用。

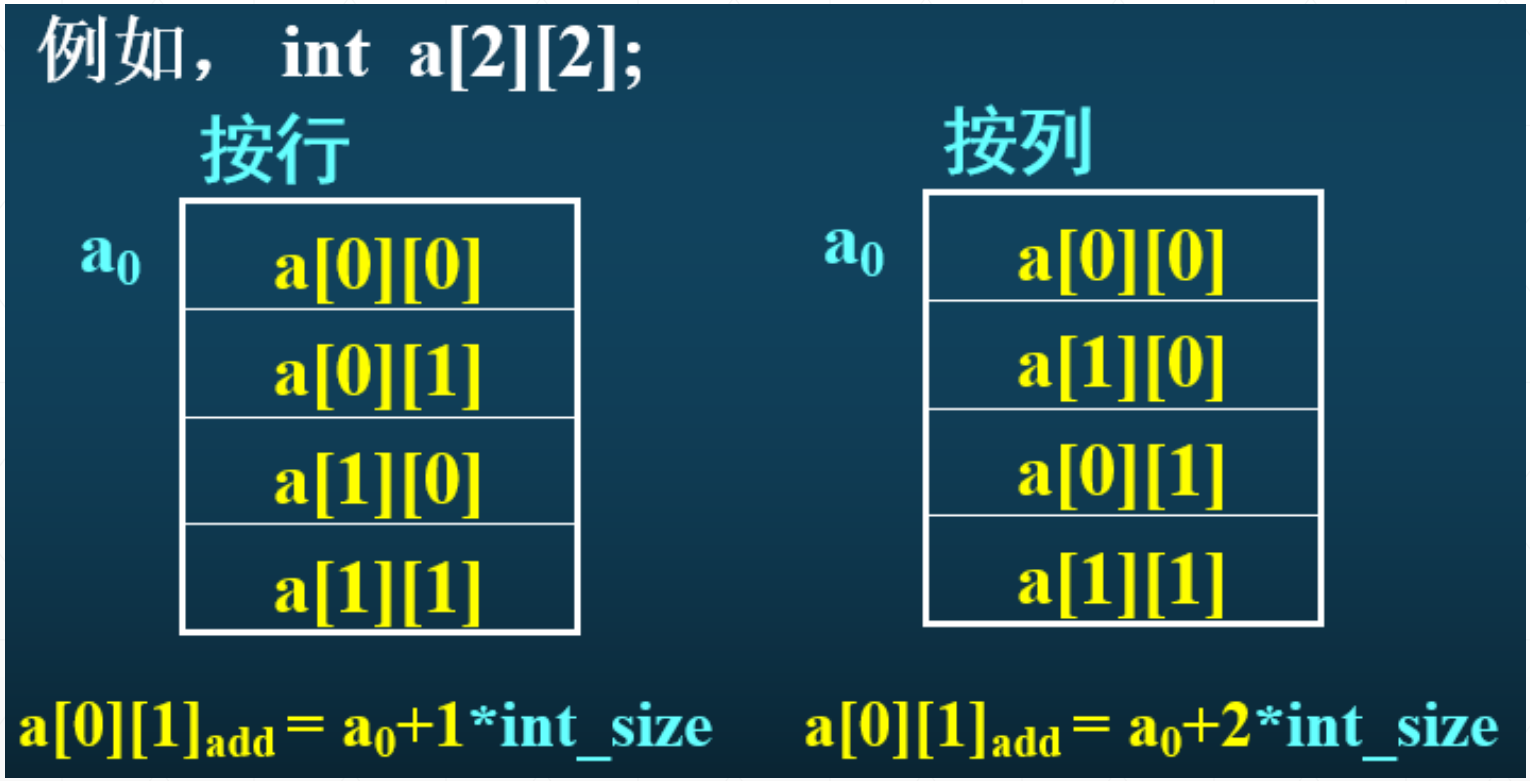
∵ A $[i_1]$ $[i_2]$… $[i_n]$ 中 i_k多为表达式

如， a[i+j-1][j++] 运行时得到下标值

数组的翻译

- 数组存储方式：按行存储、按列存储

例如， int a[2][2];



数组的翻译

- 地址计算（从0开始）

对n维数组　int a[d_1] [d_2]… [d_n];

则　a[i_1] [i_2] … [i_n] add

= a_0 + i_1 *d_2*d_3 * … *d_n

+ i_2 *d_3 *d_4*… *d_n

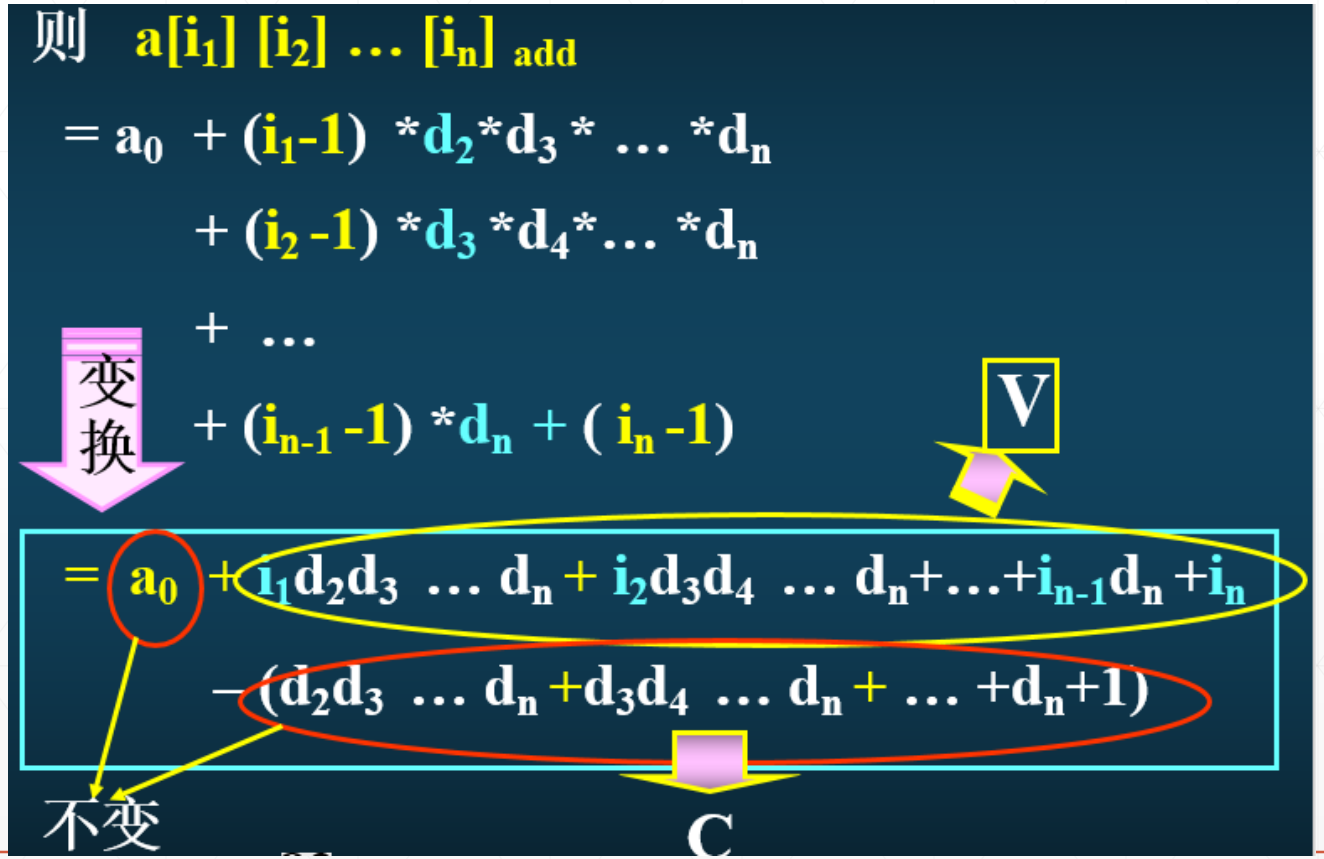
+ …

+ i_{n-1} *d_n + i_n

含i_k,是可变部分，程序运行时方可知。

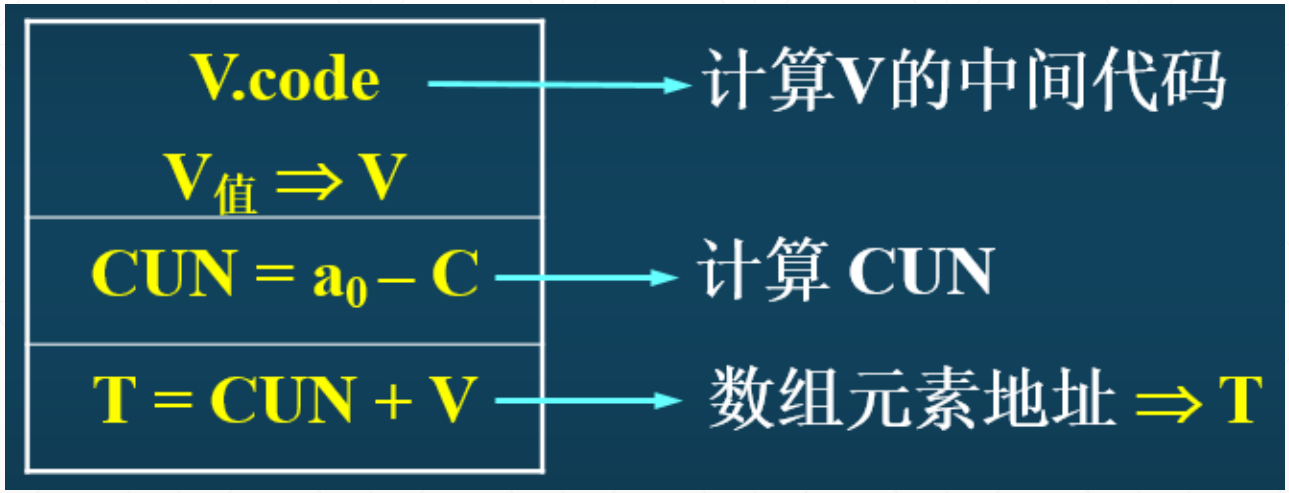
数组的翻译

- 地址计算（从1开始）



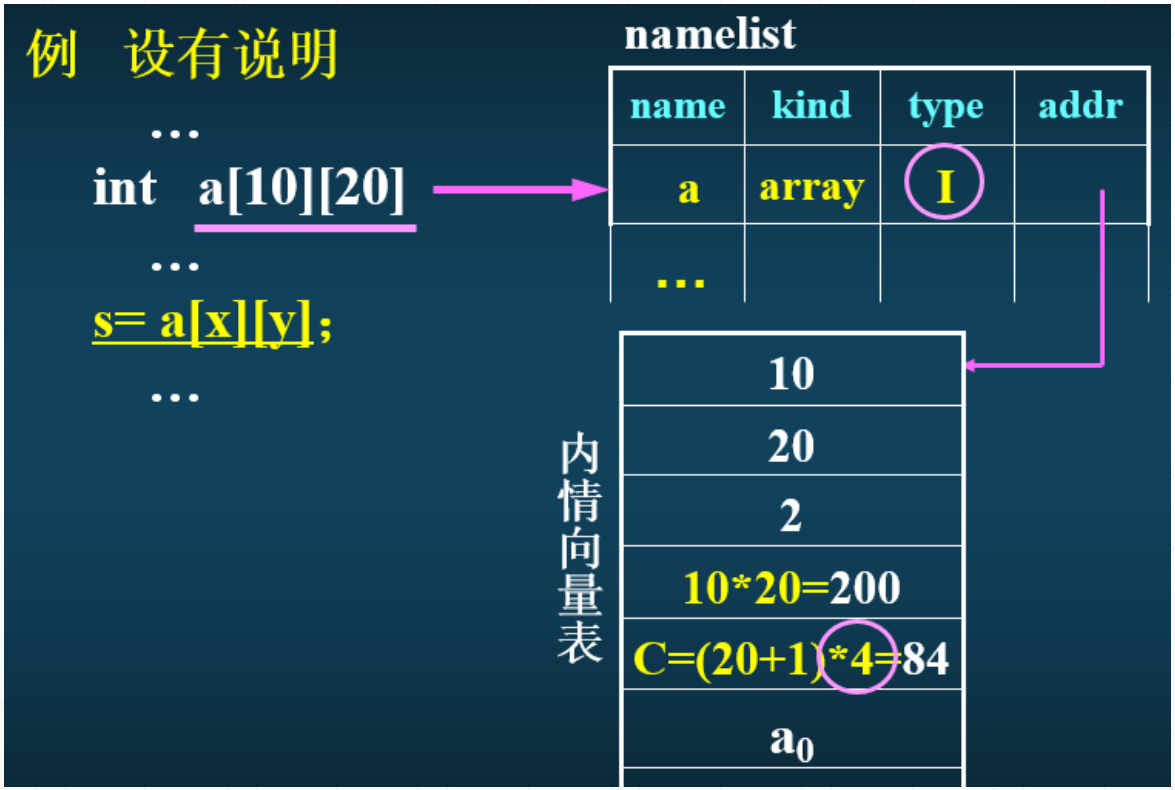
数组的翻译

- 数组元素引用目标结构



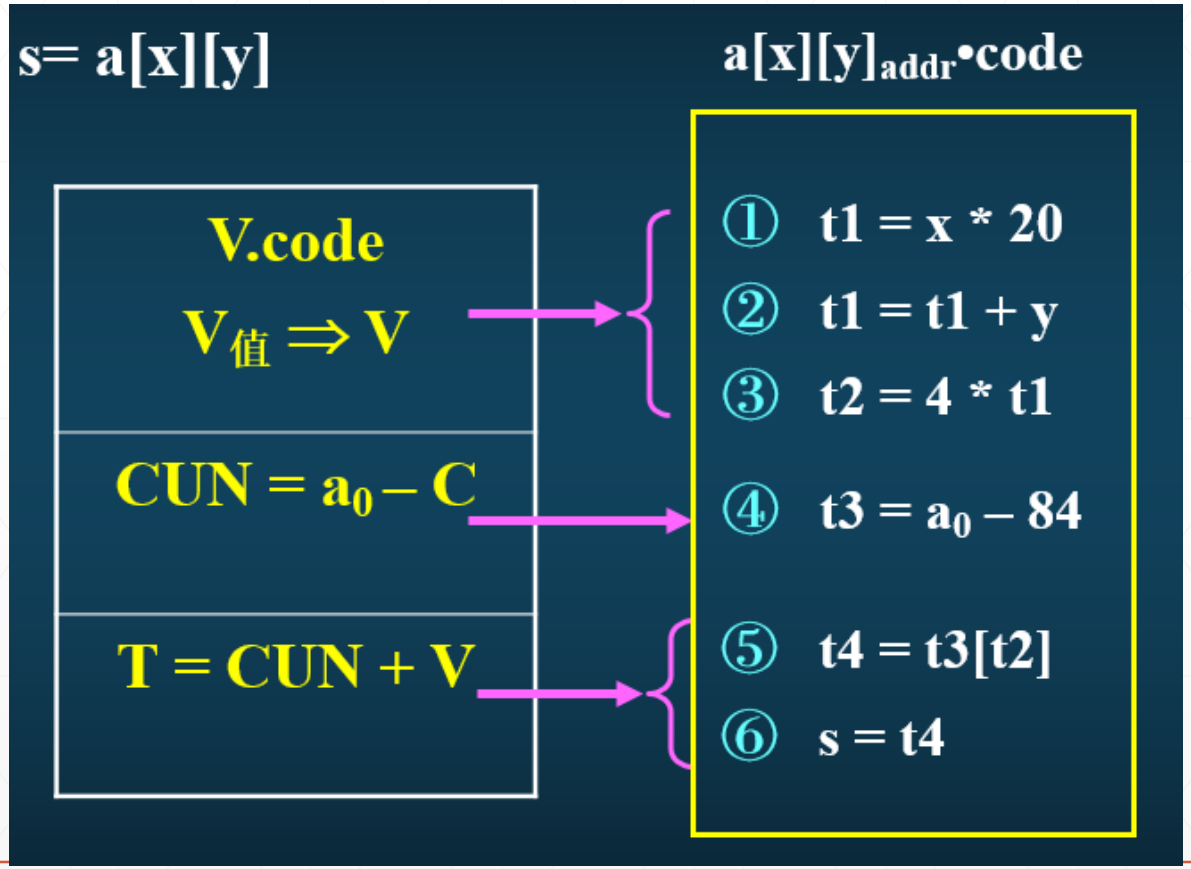
数组的翻译

- 数组元素引用目标结构



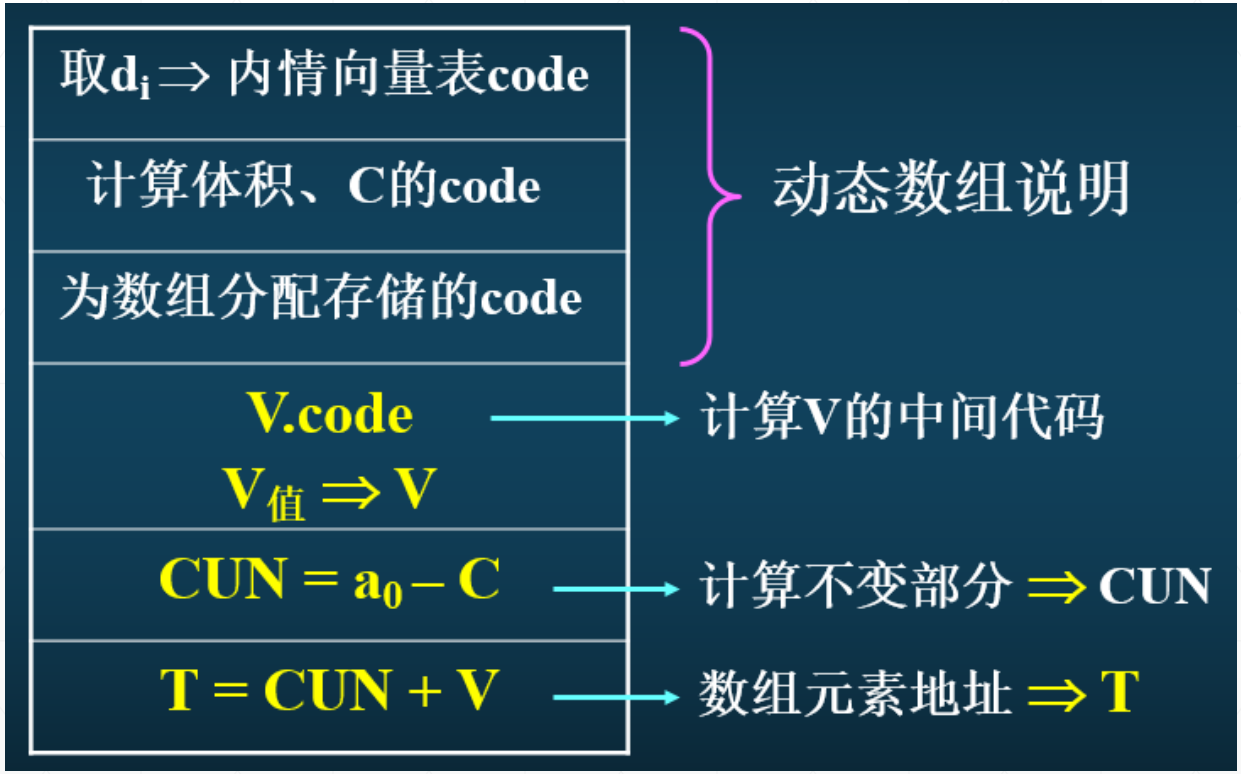
数组的翻译

- 数组元素引用目标结构



数组的翻译

- 动态数组


取d_i⇒内情向量表code
计算体积、C的code
为数组分配存储的code
动态数组说明
V.code
$V_{值}$⇒V
计算V的中间代码
CUN = a_0 – C
计算不变部分⇒CUN
T = CUN + V
数组元素地址⇒T

数组的翻译

- LLVM

```c
#include <stdio.h>

int main(void)
{
    int arr[100];

    arr[10] = 1;
    arr[50] = -1;
}

```

```
define dso_local i32 @main() #0 !dbg !14 {
  %1 = alloca [100 x i32], align 4
  %2 = getelementptr inbounds [100 x i32], ptr %1, i32 0, i32 10, !dbg !18
  store i32 1, ptr %2, align 4, !dbg !19
  %3 = getelementptr inbounds [100 x i32], ptr %1, i32 0, i32 50, !dbg !20
  store i32 -1, ptr %3, align 4, !dbg !21
  ret i32 0, !dbg !22
}
```

数组的翻译

- GCC: GIMPLE

```c
#include <stdio.h>

int main(void)
{
    int arr[100];

    arr[10] = 1;
    arr[50] = -1;
}

```

```
main ()
{
  int D.1760;

  {
    int arr[100];

    try
      {
        arr[10] = 1;
        arr[50] = -1;
      }
    finally
      {
        arr = {CLOBBER};
      }
  }
  D.1760 = 0;
  return D.1760;
}
```

函数说明和调用语句翻译

- 函数说明

函数及局部量信息登录符号表,并填入有关的属性：种属(过程或函数等)、是否为外部过程、数据类型(对函数而言)、形参个数、形参的信息(供语义检查用,如种属、类型等)、过程的入口地址等等。

函数形参的信息可以登录子表,并以某种方式和函数名的登记项连接起来.

函数说明和调用语句翻译

- 函数调用
  - 检查所调用的过程或函数是否定义；与所定义的过程或函数的类型、实参与形参的数量、顺序及类型是否一致；
  - 给被调过程或函数申请、分配活动记录所需的存储空间；
  - 计算并传送实参；
  - 加载调用结果和返回地址，恢复主调用过程或函数的继续执行；
  - 转向相应的过程或函数（转子指令）。

函数调用语句

- LLVM

```
1  #include "stdio.h"
2
3  #define SHAPE_CIRCLE    1
4  #define SHAPE_SQUARE    2
5  #define PI              3.14
6
7  float area(int shape, float r){
8      float ret;
9      switch(shape){
10         case SHAPE_CIRCLE:
11             ret = PI * r * r;
12             break;
13         case SHAPE_SQUARE:
14             ret = r * r;
15             break;
16         default:
17             ret = -1;
18     }
19     return ret;
20 }
21
22 int main(){
23     int ret;
24     ret = area(SHAPE_CIRCLE, 10);
25     ret = area(SHAPE_SQUARE, 4);
26
27     ret;
28 }
```

```
40
41 define dso_local i32 @main() #0 !dbg !36 {
42   %1 = alloca i32, align 4
43   %2 = call float @area(i32 noundef 1, float noundef 1.000000e+01), !dbg !37
44   %3 = fptosi float %2 to i32, !dbg !37
45   store i32 %3, ptr %1, align 4, !dbg !38
46   %4 = call float @area(i32 noundef 2, float noundef 4.000000e+00), !dbg !39
47   %5 = fptosi float %4 to i32, !dbg !39
48   store i32 %5, ptr %1, align 4, !dbg !40
49   %6 = load i32, ptr %1, align 4, !dbg !41
50   ret i32 0, !dbg !42
51 }
52
```

函数调用语句

- GCC: GIMPLE

```c
#include "stdio.h"

#define SHAPE_CIRCLE    1
#define SHAPE_SQUARE    2
#define PI              3.14

float area(int shape, float r){
    float ret;
    switch(shape){
        case SHAPE_CIRCLE:
            ret = PI * r * r;
            break;
        case SHAPE_SQUARE:
            ret = r * r;
            break;
        default:
            ret = -1;
    }
    return ret;
}

int main(){
    int ret;
    ret = area(SHAPE_CIRCLE, 10);
    ret = area(SHAPE_SQUARE, 4);

    ret;
}
```

```
main ()
{
  float D.2227;
  float D.2228;
  int D.2229;

  {
    int ret;

    D.2227 = area (1, 1.0e+1);
    ret = (int) D.2227;
    D.2228 = area (2, 4.0e+0);
    ret = (int) D.2228;
  }
  D.2229 = 0;
  return D.2229;
}
```