Compilation Principle
编 译 原 理

第17讲：中间代码(2)

张献伟

xianweiz.github.io

DCS290, 05/13/2021

Review Questions (1)

- What are the main tasks of compilation backend?

  Intermediate code generation, optimizations, target code generation

- What is IR (specifically, the low-level IR)?

  Intermediate Representation. A machine- and language-independent version of the original source code.

- Why do we use IR?

  Clean separation of front- and back-end; easy to optimize and extend

- What is three-address code (TAC)?

  A type of IR, with at most three operands. (High-level assembly)

- TAC of x + y * z + 5?

  $t_1 = y * z$; $t_2 = x + t_1$; $t_3 = t_2 + 5$;

SUN YAT-SEN UNIVERSITY

Review Questions (2)

- Possible ways to implement TAC?

  Quadruples: op arg1, arg2, result
  Triples: op arg1 arg2
  Indirect triples: op arg1 arg2

- What is Single Static Assignment?

  An IR that facilitates certain code optimization.
  Give variable different version name on every assignment.

- In code generation, how to layout variables in memory?

  Calculate the location using base address and type width.

- What is address alignment?

  Enforce addr(x) % sizeof(x.type) == 0, to respect the hardware constraints to avoid unexpected performance degradation.

中山大學 SUN YAT-SEN UNIVERSITY

Code Generation [代码生成]

- To generate three-address codes (TACs)
  - By now, we have
    - an AST, annotated with scope and type information
  - Do another round of tree traversal
    - Lay out variables in memory
    - Generate TAC for any subexpressions or substatements
    - Using the result, generate TAC for the overall expression
- We will use the syntax-directed formalisms to specify translation
  - Variable definitions [变量定义]
  - Assignment [赋值]
  - Array references [数组引用]
  - Boolean expressions [布尔表达式]
  - Control-flow statements [控制流语句]

Type Expressions [类型表达式]

- A **type expression** is either a basic type or is formed by applying an operator called a *type constructor* [类型构造符] to a type expression
  - Basic type: *integer*, *float*, *char*, *Boolean*, *void*
  - Array: *array(I, T)* is a type expression, if *T* is
    - int[3] <--> array(3, int)
    - int[2][3] <--> array(2, array(3, int))
  - Pointer: *pointer(T)* is a type expression, if *T* is
    - int *val <--> pointer(int)

P -> *D*
D -> *T* id; D_1 | ε
T -> *B* *C* | *T_1
B -> int | real
C -> [num]C_1 | ε

CodeGen: Variable Definitions

- Translating variable definitions
  - *enter(name, type, offset)*
    - Save the type and relative address in the symbol-table entry for the name

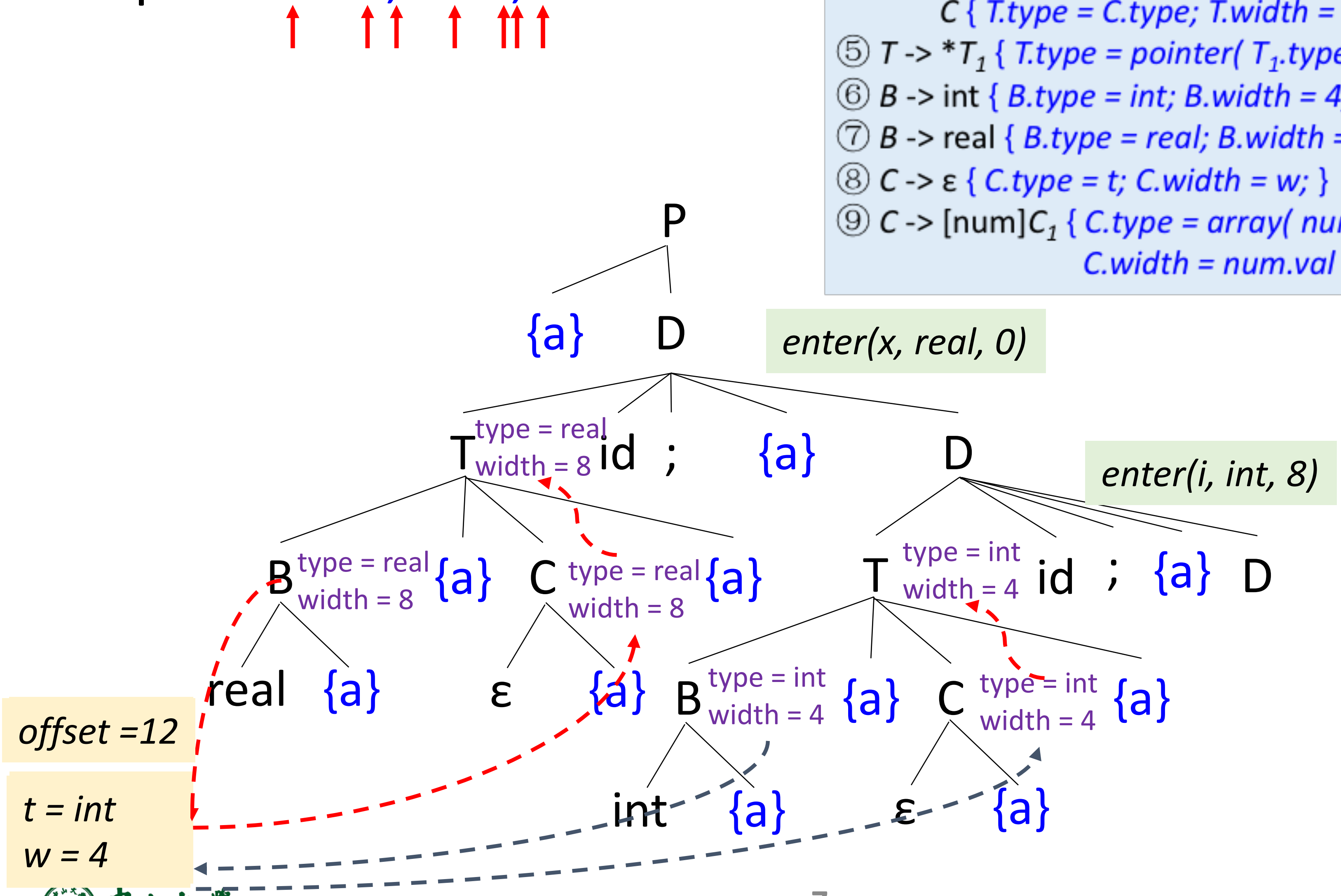
```
① P -> { offset = 0 } D
② D -> T id; { enter( id.lexeme, T.type, offset );
                  offset = offset + T.width; } D₁
③ D -> ε
④ T -> B { t = B.type; w = B.width; }
         C { T.type = C.type; T.width = C.width; }
⑤ T -> *T₁ { T.type = pointer( T₁.type); T.width = 4; }
⑥ B -> int { B.type = int; B.width = 4; }
⑦ B -> real { B.type = real; B.width = 8; }
⑧ C -> ε { C.type = t; C.width = w; }
⑨ C -> [num]C₁ { C.type = array( num.val, C₁.type);
                      C.width = num.val * C₁.width; }
```

- Examples:
  - *real x; int i;*
  - *int[2][3];*
- *type, width*
  - Syn attributes
- *t, w*
  - Vars to pass type and width from B node to the node for *C* -> ε
- *offset*
  - The next relative address

Example

- Input: real x; int i;

① $P \to \{ offset = 0 \}\ D$
② $D \to T$ id; $\{ enter(id.lexeme, T.type, offset);$
$offset = offset + T.width; \}\ D_1$
③ $D \to \varepsilon$
④ $T \to B\ \{ t = B.type; w = B.width; \}$
$C\ \{ T.type = C.type; T.width = C.width; \}$
⑤ $T \to *T_1\ \{ T.type = pointer(T_1.type); T.width = 4; \}$
⑥ $B \to$ int $\{ B.type = int; B.width = 4; \}$
⑦ $B \to$ real $\{ B.type = real; B.width = 8; \}$
⑧ $C \to \varepsilon\ \{ C.type = t; C.width = w; \}$
⑨ $C \to$ [num]$C_1\ \{ C.type = array(num.val, C_1.type);$
$C.width = num.val * C_1.width; \}$



CodeGen: Assignment Statement

- Translate into *three-address code* [赋值语句]
  - An expression with more than one operator will be translated into instructions with at most one operator per instruction
- Helper functions in translation
  - *lookup (id)*: search *id* in symbol table, return null if none
  - *emit()/gen()*: generate three-address IR
  - *newtemp()*: get a new temporary location

① S -> id = E;
② E -> E_1 + E_2;
③ E -> - E_1
④ E -> (E_1)
⑤ E -> id

Assignment statement:
a = b + (-c)

Three-address code:
t_1 = minus c
t_2 = b + t_1
a = t_2

SDT Translation of Assignment

- Attributes ***code*** and ***addr***
  - *S.code* and *E.code* denote the TAC for *S* and *E*, respectively
  - *E.addr* denotes the address that will hold the value of *E* (can be a name, constant, or a compiler-generated temporary)

```
① S -> id = E; { p = lookup(id.lexeme); if !p then error;
                  S.code = E.code ||
                  gen( p '=' E.addr ); }
② E -> E1 + E2; { E.addr = newtemp();
                   E.code = E1.code || E2.code ||
                   gen(E.addr '=' E1.addr '+' E2.addr); }
③ E -> - E1 { E.addr = newtemp();
              E.code = E1.code ||
              gen(E.addr '=' 'minus' E1.addr); }
④ E -> (E1) { E.addr = E1.addr;
              E.code = E1.code; }
⑤ E -> id { E.addr = lookup(id.lexeme); if !E.addr then error;
            E.code = ''; }
```

Incremental Translation[增量翻译]

- Generate only the new three-address instructions
  - *gen()* not only constructs a three-address inst, it appends the inst to the sequence of insts generated so far

① S -> id = E; { *p = lookup(*id.*lexeme); if !p then error;*

gen(p '=' E.addr); }

② E -> E_1 + E_2; { *E.addr = newtemp();*

gen(E.addr '=' E_1.addr '+' E_2.addr); }

③ E -> - E_1 { *E.addr = newtemp();*

gen(E.addr '=' 'minus' E_1.addr); }

④ E -> (E_1) { *E.addr = E_1.addr;*

}

⑤ E -> id { *E.addr = lookup(*id.*lexeme); if !E.addr then error;*

}

Code attributes can be long strings

Example

① S -> id = E; { *p = lookup(id.lexeme); if !p then error;*
gen(p '=' E.addr); }
② E -> $E_1 + E_2$; { *E.addr = newtemp();*
gen(E.addr '=' E_1.addr '+' E_2.addr); }
③ E -> - E_1 { *E.addr = newtemp();*
gen(E.addr '=' 'minus' E_1.addr); }
④ E -> (E_1) { *E.addr = E_1.addr; }*
⑤ E -> id { *E.addr = lookup(id.lexeme); if !E.addr then error; }*

- Input

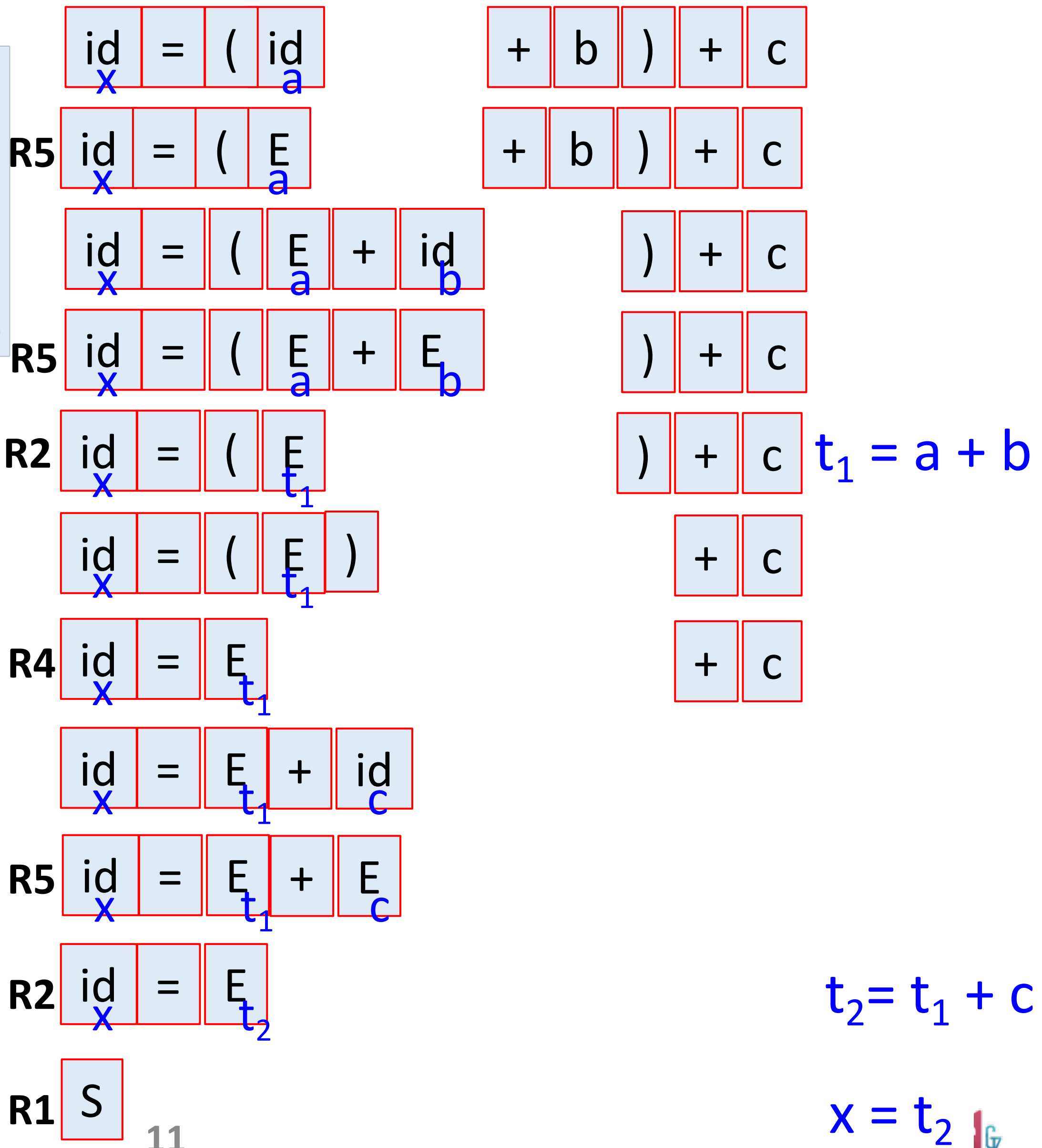
  x = (a + b) + c

- Translated TAC

  $t_1 = a + b$

  $t_2 = t_1 + c$

  $x = t_2$

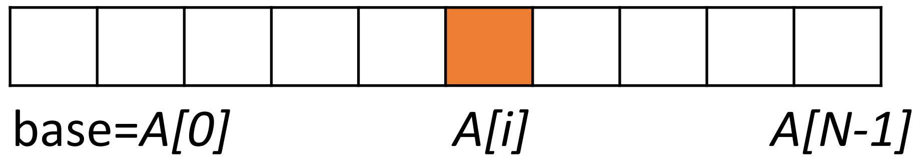


CodeGen: Array Reference[数组引用]

- Primary problem in generating code for array references is to *determine the address of element*
- 1D array

  int A[N];

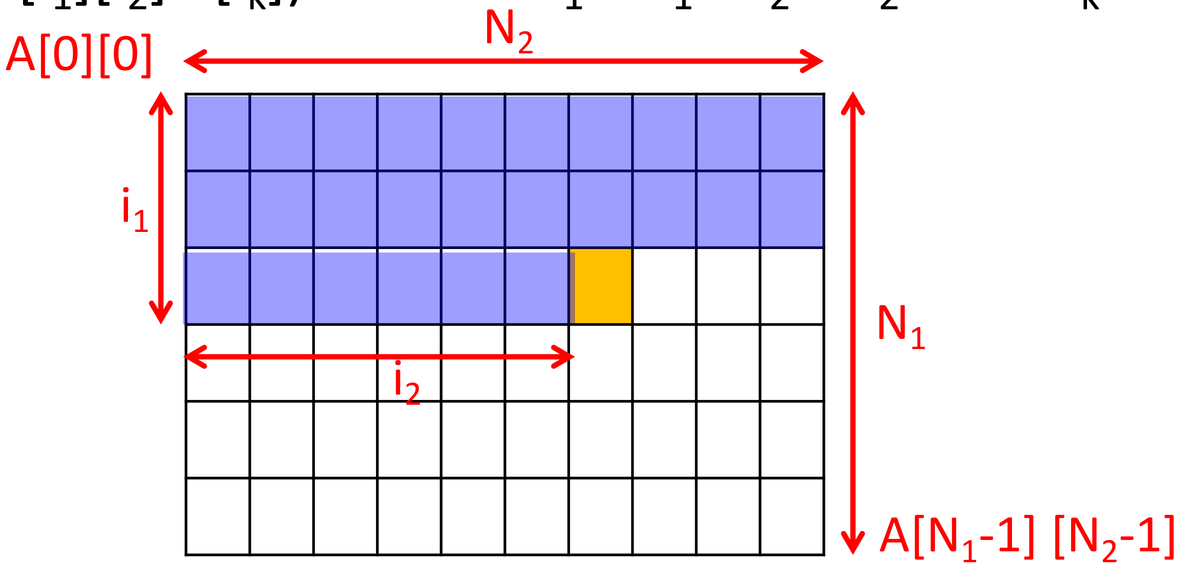
  A[i] ++;



  - *Base*: address of the first element
  - *Width*: width of each element
    - *i*×*width* is the offset
- Addressing an array element
  - addr(A[i]) = base + i×width

N-dimensional Array

- Laying out 2D array in 1D memory
  - *int A[N_1][N_2]; /** int A[0..N_1][0..N_2] */
  - *A[i_1][i_2] ++;*
- The organization can be row-major or column-major
  - C language uses row major (i.e., stored row by row)
  - Row-major: $\text{addr}(A[i_1, i_2]) = \text{base} + (i_1 \times \underbrace{N_2 * \text{width}}_{w_1} + i_2 \times \underbrace{\text{width}}_{w_2})$
- *k*-dimensional array
  - $\text{addr}(A[i_1][i_2]\ldots[i_k]) = \text{base} + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$



Translation of Array References

- Type(a) = array(10, int)
  - c = a[i];

addr(a[i]) = base + i*4

t_1 = i * 4
t_2 = a[t_1]
c = t_2

addr(a[i_1][i_2]) = base + i_1*20 + i_2*4

- Type(a) = array(3, array(5, int))
  - c = a[i_1][i_2];

t_1 = i_1 * 20
t_2 = i_2 * 4
t_3 = t_1 + t_2
t_4 = a[t_3]
c = t_4

- Type(a) = array(3, array(5, array(8, int)))
  - c = a[i_1][i_2][i_3]

addr(a[i_1][i_2][i_3]) = base + i_1*w_1 + i_2*w_2 + i_3*w_3
= base + i_1*160 + i_2*32 + i_3*4

SUN YAT-SEN UNIVERSITY

Translation of Array References (cont.)

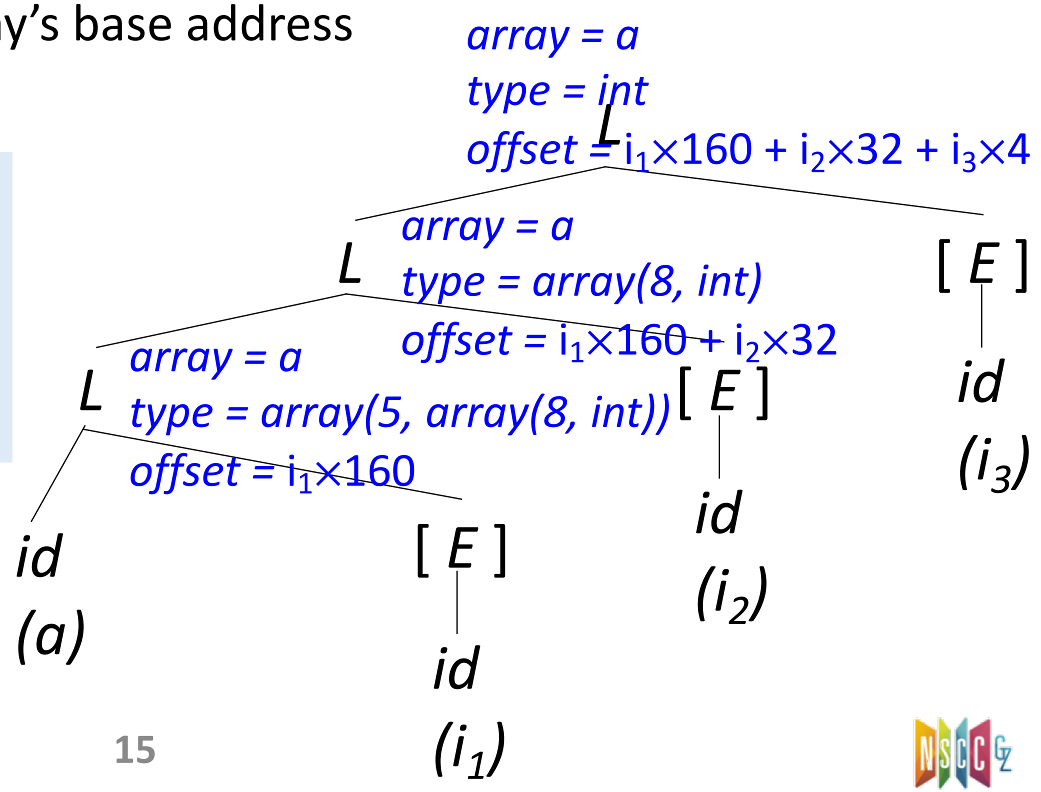
- $A[i_1][i_2][i_3]$, type(a) = array(3, $\underline{\text{array(5, array(8, int)}}$))
  - *L.type*: the type of the subarray generated by *L*
  - *L.addr*: a temporary that is used while computing the offset for the array reference by summing the terms $i_j \times w_j$
  - *L.array*: a pointer to the symbol-table entry for the array name
    - *L.array.base* gives the array's base address

① *S* -> id = *E*; | *L = E;*

② *E* -> $E_1 + E_2$ | - E_1 | (E_1) | id | *L*

③ *L* -> id [*E*] | L_1 [*E*]

$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$



array = a
type = int
offset = $i_1 \times 160 + i_2 \times 32 + i_3 \times 4$
L
array = a
type = array(8, int)
offset = $i_1 \times 160 + i_2 \times 32$
L
[E]
array = a
type = array(5, array(8, int))
offset = $i_1 \times 160$
L
[E]
id
(i_3)
id
(i_2)
id
(a)
[E]
id
(i_1)

Translation of Array References (cont.)

- A[i_1][i_2][i_3], type(a) = array(3, array(5, array(8, int)))

```
① S -> id = E; | L = E; { gen(L.array.base’[‘L.addr’]’ ‘=‘ E.addr); }
② E -> E1 + E2 | - E1 | (E1) | id | L { E.addr = newtemp();
                             gen(E.addr ’=‘ L.array.base’[‘L.addr’]’); }
③ L -> id [E] { L.array = lookup(id.lexeme); if !L.array then error;
                          L.type = L.array.type.elem;
                          L.offset = newtemp();
                          gen(L.addr ‘=‘ E.addr ‘*’ L.type.width); }
     | L1 [E] { L.array = L1.array;
                     L.type = L1.type.elem;
                     t = newtemp();
                     gen(t ‘=‘ E.addr ‘*’ L.type.width);
                     L.addr = newtemp();
                     gen(L.addr ‘=‘ L1.addr ‘+’ t; }
```

$t_1 = i_1 * 160$
$t_2 = i_2 * 32$
$t_3 = t_1 + t_2$
$t_4 = i_3 * 4$
$t_5 = t_3 + t_4$
$c = a[t_5]$

CodeGen: Boolean Expressions

- Boolean expression: a *op* b
  - where op can be <, <=, = !=, > or >=, &&, ||, …
- **Short-circuit** evaluation[短路计算]: to skip evaluation of the rest of a boolean expression once a boolean value is known
  - Given following C code: *if (flag || foo()) { bar(); };*
    - If *flag* is true, *foo()* never executes
    - Equivalent to: *if (flag) { bar(); } else if (foo()) { bar(); };*
  - Given following C code: *if (flag && foo()) { bar(); };*
    - If *flag* is false, *foo()* never executes
    - Equivalent to: *if (!flag) { } else if (foo()) { bar(); };*
  - Used to alter control flow, or compute logical values
    - Examples: *if (x < 5) x = 1*; *x = true*; *x = a < b*
    - For control flow, boolean operators translate to ***jump*** statements

Boolean Exprs (w/o Short-Circuiting)

- Computed just like any other arithmetic expression

E -> (a < b) or (c < d and e < f)

```
t1 = a < b
t2 = c < d
t3 = e < f
t4 = t2 && t3
t5 = t1 || t4
```

- Then, used in control-flow statements
  - *S.next*: label for code generated after *S*

S -> if E S1

```
if (!t5) goto S.next
S1.code
S.next: ...
```