Compilation Principle
编 译 原 理

第18讲：中间代码(2)

张献伟

xianweiz.github.io

DCS290, 5/12/2022

Review Questions

- 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?

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

- Possible ways to implement TAC?

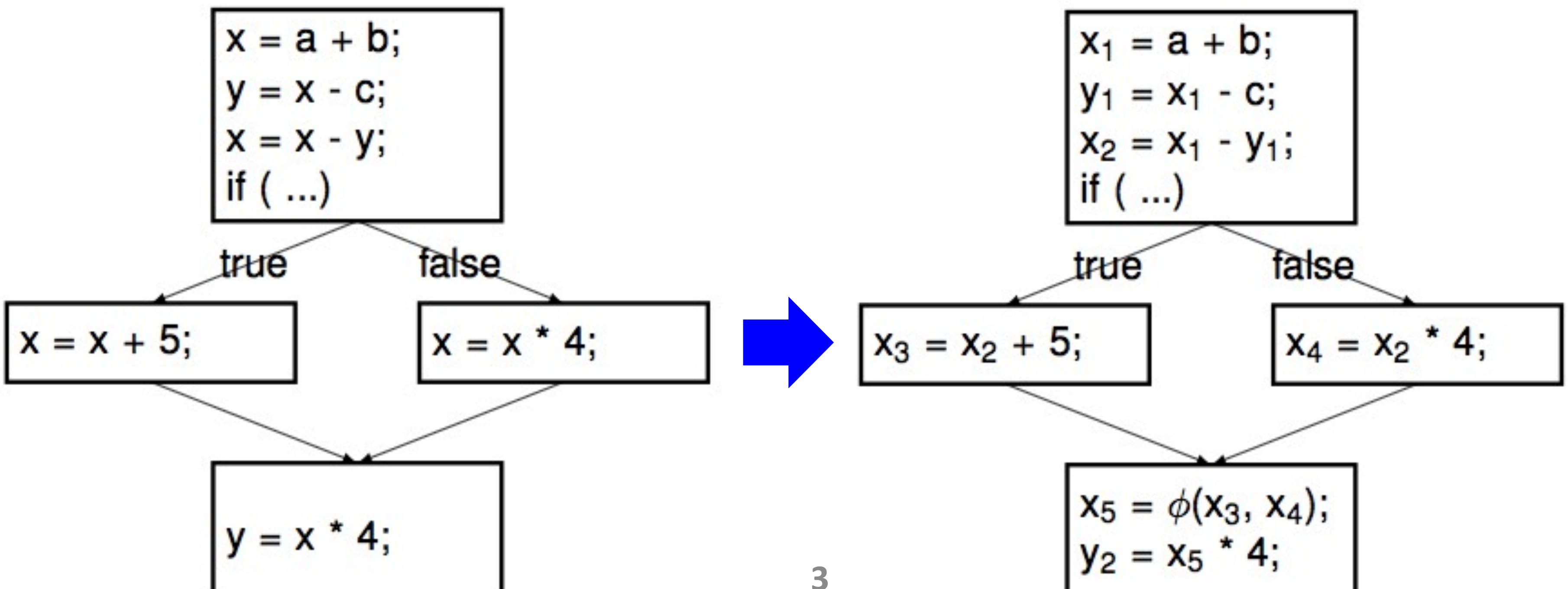
  Quadruples: op arg1, arg2, result

  Triples: op arg1 arg2

  Indirect triples: op arg1 arg2

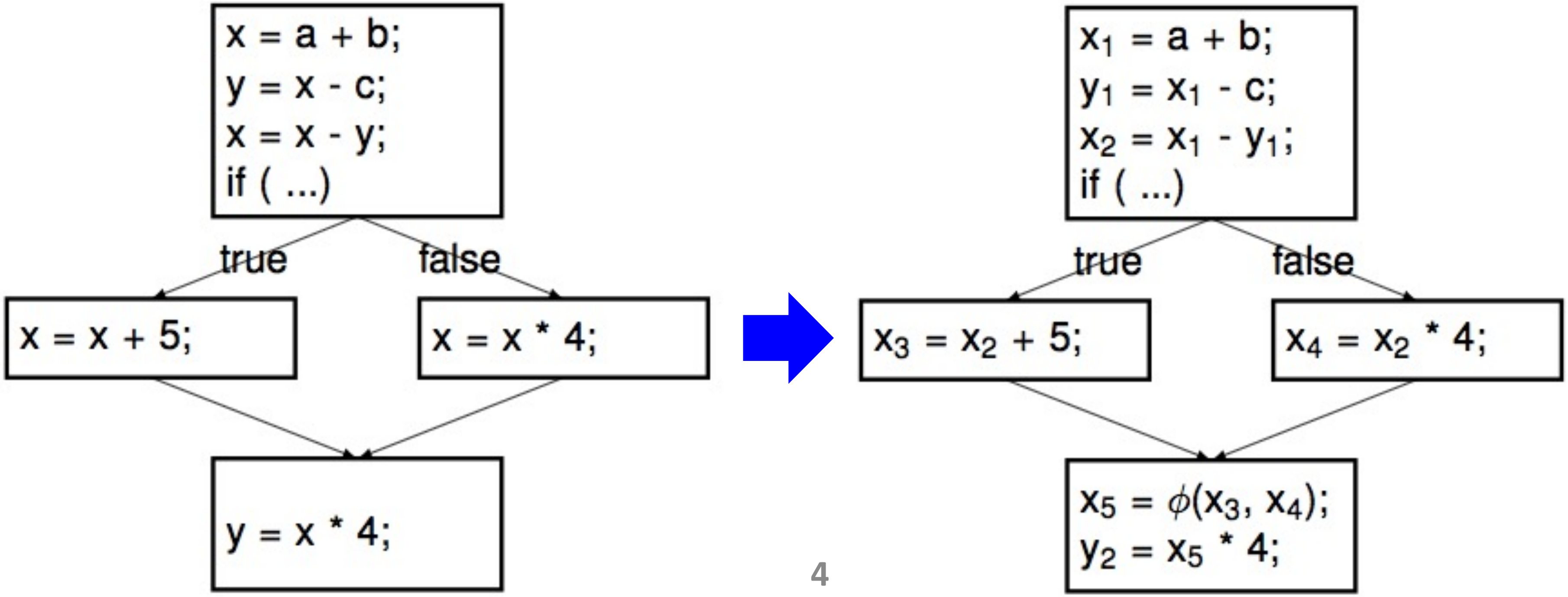
Single Static Assignment[静态单赋值]

- Every variable is assigned to exactly once statically[仅一次]
  - Give variable a different version name on every assignment
    - e.g. $x \rightarrow x_1, x_2, \ldots, x_5$ for each static assignment of x
  - Now value of each variable guaranteed not to change
  - On a control flow merge, ϕ-function combines two versions
    - e.g. $x_5 = \phi(x_3, x_4)$: means x_5 is either x_3 or x_4



Benefits of SSA

- SSA is an IR that facilitates certain code optimizations
  - SSA tells you when an optimization shouldn't happen
  - Suppose compiler performs CSE on previous example:
    - Without SSA, (incorrectly) tempted to eliminate second x * 4
      - x = x * 4; y = x * 4; ➔ x = x * 4; y = x;
    - With SSA, x_2 * 4 and x_5 * 4 are clearly different values



Benefits of SSA (cont.)

- SSA is an IR that facilitates certain code optimizations
  - SSA tells you when an optimization should happen
  - Suppose compiler performs dead code elimination (DCE): (DCE removes code that computes dead values)

Before	After
x = a + b;	x_1 = a + b;
x = c - d;	x_2 = c - d;
y = x * b;	y_1 = x_2 * b;

  - Without SSA, not very clear whether there are dead values
  - With SSA, x_1 is never used and clearly a dead value
- Why does SSA work so well with compiler optimizations?
  - SSA makes flow of values explicit in the IR
  - Without SSA, need a separate dataflow graph
  - Will discuss more in **Compiler Optimization** section

SSA Orthogonal to IR Impl.[正交关系]

- SSA is expressed most commonly as TACs
- We learned 3 ways to implement TACs
  - quadruples
  - triples
  - indirect triples
- How you implement is orthogonal to SSA representation
  - After variable renaming, any 3-address code becomes SSA
- SSA is used widely in modern compilers:
  - GCC (GNU C Compiler)
  - LLVM Compiler
  - Oracle Java JIT Compiler
  - Google Chrome JavaScript JIT Compiler
  - PyPy Python JIT Compiler

LLVM: SSA and Phi

- All LLVM instructions are represented in the Static Single Assignment (SSA) form
  - Affable to the design of simpler algorithms for existing optimizations and has facilitated the development of new ones
- The ‘phi’ instruction is used to implement the ϕ node in the SSA graph representing the function
  - <result> = phi [fast-math-flags] <ty> [<val0>, <label0>], ...
  - At runtime, the ‘phi’ instruction logically takes on the value specified by the pair corresponding to the predecessor basic block that executed just prior to the current block

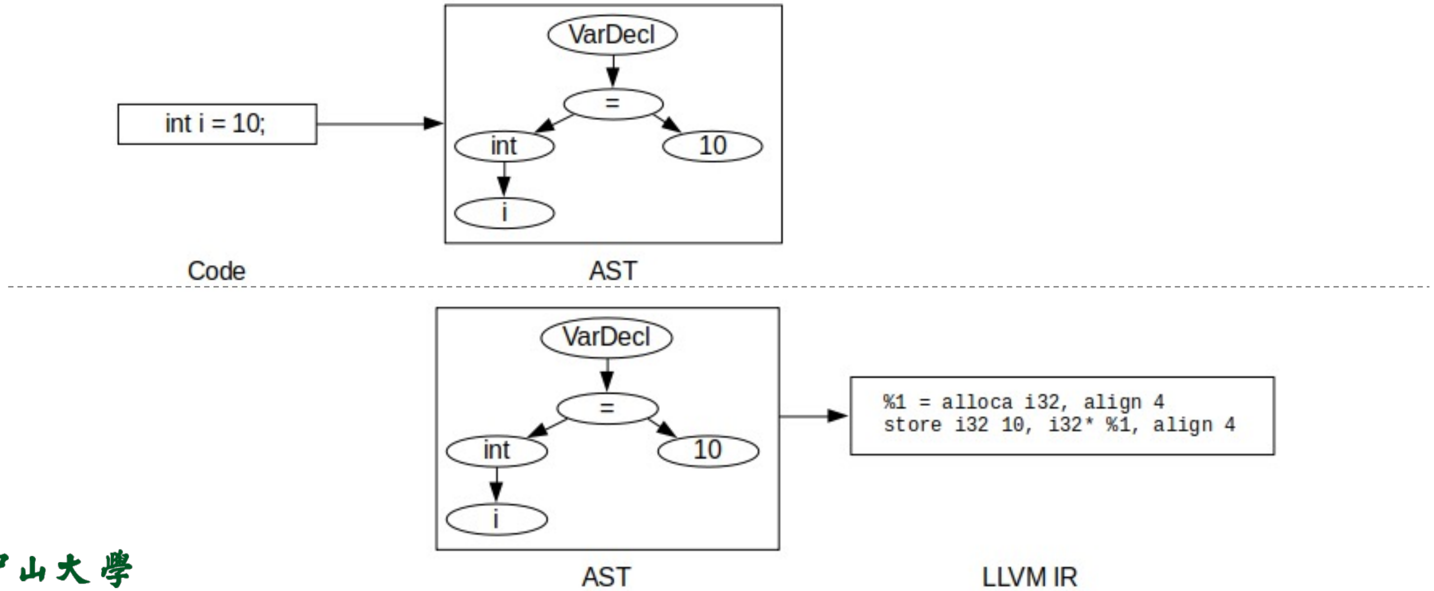
```
a = 1;
if (v < 10)
   a = 2;
b = a;
```

a_1 = 1;
if (v < 10)
   a_2 = 2;
b = PHI(a_1, a_2);

https://llvm.org/docs/LangRef.html#phi-instruction
https://www.llvmpy.org/llvmpy-doc/dev/doc/llvm_concepts.html#ssa-form-and-phi-nodes

Generating Code: AST to IR[IR生成]

- By now, we have
  - An AST, annotated with scope and type information
- To generate three-address codes (TACs)
  - Traversing the AST after the parse[单独遍历]
    - Writing a codeGen method for the appropriate kinds of AST nodes
  - Syntax-directed translation[语法制导]
    - Generating code while parsing



http://freecompilercamp.org/clang-basics/

Syntax Directed Translation[语法制导翻译]

- Syntax directed translation can be used again for code generation[代码生成]
  - Since code generation is also dependent on syntax/AST
  - Code generation is translating syntactic structures to code
- What language structures do we need to translate?[翻译]
  - Definitions (variables, functions, ...)
  - Assignment statements
  - Control flow statements (if-then-else, for-loop, ...)
  - ...
- We are going to use the following strategy:
  - Specify <u>SDD semantic rules</u> (without ordering)
  - Convert SDD rules to <u>SDT actions</u> (with ordering)
    - In the process, we will discover SDD has non-*L-attributes*
    - We will also discuss what to do with those non-*L-attributes*

Code Generation Overview[代码生成]

- Program code is a collection of functions
  - By now, all functions are listed in symbol table
- Goal is to generate code for each function in that list
- Generating code for a function involves two steps:
  - Processing variable definitions[变量定义]
    - Involves laying out variables in memory
  - Processing statements[语句]
    - Involves generating instructions for statements
      - Assignment[赋值]
      - Array references[数组引用]
      - Boolean expressions[布尔表达式]
      - Control-flow statements[控制流语句]
      - …
- We will start with <u>processing variable definitions</u>

Processing Variable Definitions[变量定义]

- To lay out a variable, both location and width are needed
  - Location: where variable is located in memory
  - Width: how much space variable takes up in memory
- Attributes for variable definition:
  - **T V**   e.g. int x;
  - **T**: non-terminal for type name
    - **T.type**: type (int, float, ...)
    - **T.width**: width of type in bytes (e.g. 4 for int)
  - **V**: non-terminal for variable name
    - **V.type**: type (int, float, ...)
    - **V.width**: width of variable according to type
    - **V.offset**: offset of variable in memory
  - But offset from what...?

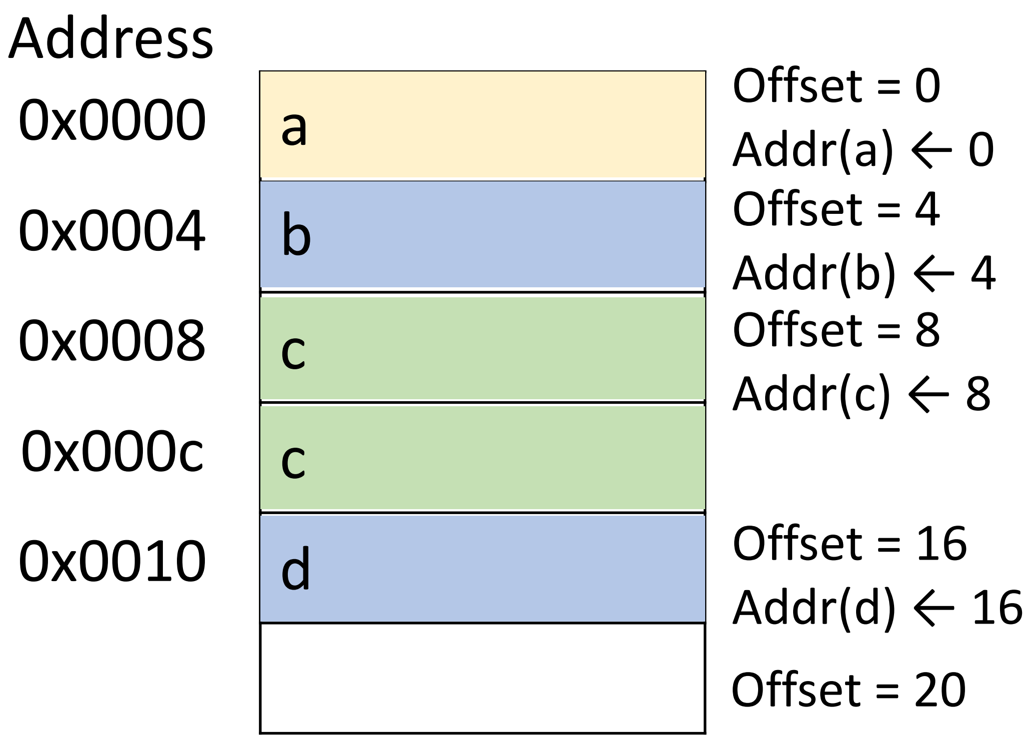
Calculate Variable Location from Offset

- Naive method: reserve a big memory section for all data
  - Size data section to be large enough to contain all variables
  - Location = var offset + base of data section
- Naive method wastes a lot of memory
  - Vars with limited scope need to live only briefly in memory
    - E.g. function variables need to last only for duration of call
- **Solution**: allocate memory briefly for each scope[域内]
  - Allocate when entering scope, free when exiting scope
  - Variables in the same scope are allocated / freed together
  - Location = var offset + base of scope memory section
  - Will discuss more later in **Runtime Management**

Storage Layout of Variables in a Function

- When there are multiple variables defined in a function,
  - Compiler lays out variables in memory <u>sequentially</u>
  - Current <u>offset</u> used to place variable x in memory
    - address(x) ← offset
    - offset += sizeof(x.type)

```
void foo() {
  int a;
  int b;
  long long c;
  int d;
}
```



More about Storage Layout

- Allocation alignment[对齐]
  - Enforce addr(x) % sizeof(x.type) == 0
  - Most machine architectures are designed such that computation is most efficient at sizeof(x.type) boundaries
    - E.g. most machines are designed to load integer values at integer word boundaries
    - If not on word boundary, need to load two words and shift & concatenate → inefficient

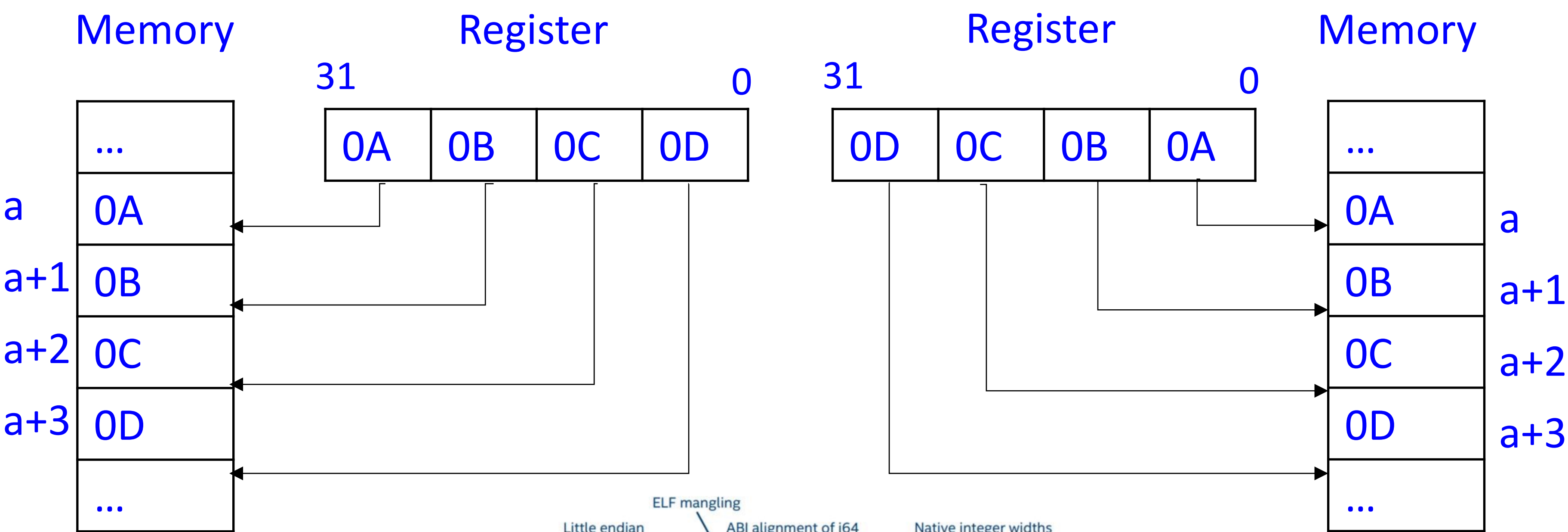
```
void foo() {
  char a;         // addr(a) = 0
  int b;          // addr(b) = 1
  int c;          // addr(c) = 5
  long long d;    // addr(d) = 9
}
```

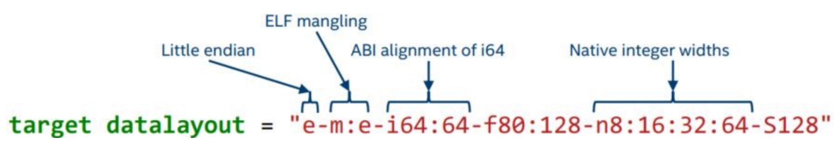
→

```
void foo() {
  char a;         // addr(a) = 0
  int b;          // addr(b) = 4
  int c;          // addr(c) = 8
  long long d;    // addr(d) = 16
}
```

More about Storage Layout (cont.)

- Endianness[字节序]
  - Big endian: MSB (most significant byte) in lowest address
  - Little endian: LSB (least significant byte) in lowest address





Big-endian[大字节序] Little-endian[小字节序]

https://www.geeksforgeeks.org/little-and-big-endian-mystery/

Type Expressions[类型表达式]

- A **type expression** is either a basic type or is formed by applying an operator called a <u>*type constructor*</u>[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;* } D_1
③ *D* -> ε
④ *T* -> *B* { *t = B.type; w = B.width;* } *C* { *T.type = C.type; T.width = C.width;* }
⑤ *T* -> *T_1 { *T.type = pointer(* T_1*.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]C_1 { *C.type = array(num.val,* C_1*.type); C.width = num.val ** C_1*.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 -> { $offset = 0$ } D

② D -> T id; { $enter(id.lexeme, T.type, offset)$; $offset = offset + T.width$; } D_1

③ D -> ε

④ T -> B { $t = B.type$; $w = B.width$; } C { $T.type = C.type$; $T.width = C.width$; }

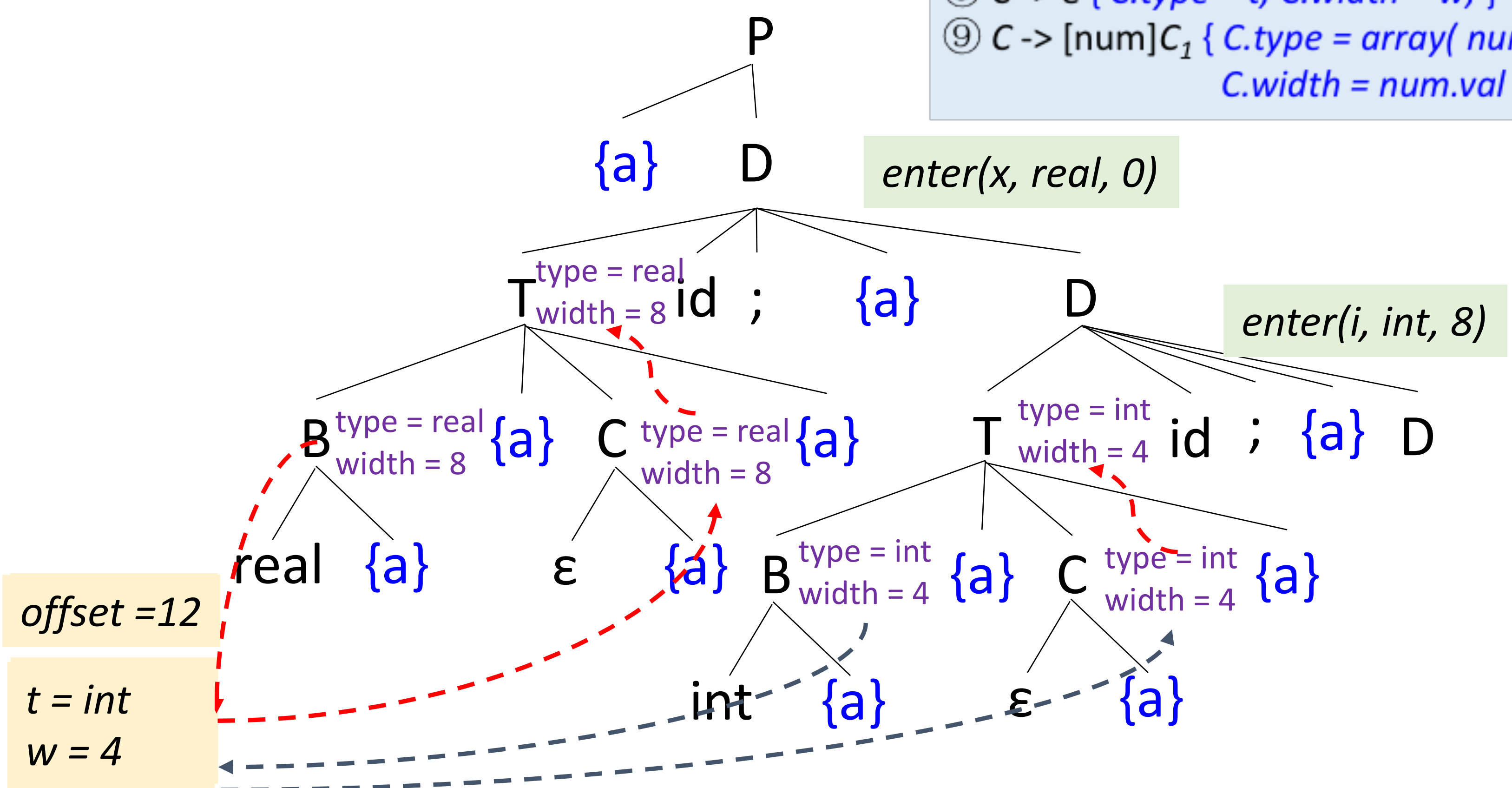
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type)$; $T.width = 4$; }

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]C_1 { $C.type = array(num.val, C_1.type)$; $C.width = num.val * C_1.width$; }



Code Generation[代码生成]

- 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]

- To generate three-address codes (TACs)
  - Lay out variables in memory
  - Generate TAC for any subexpressions or substatements
  - Using the result, generate TAC for the overall expression

CodeGen: Assignment Statement

- Translate into ***<u>three-address code</u>***[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* -> E_1 + E_2; { *E.addr = newtemp();*
E.code = E_1*.code ||* E_2*.code ||*
gen(E.addr '=' E_1*.addr '+'* E_2*.addr);* }

③ *E* -> - E_1 { *E.addr = newtemp();*
E.code = E_1*.code ||*
gen(E.addr '=' 'minus' E_1*.addr);* }

④ *E* -> (E_1) { *E.addr =* E_1*.addr;*
E.code = E_1*.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

中山大学 SUN YAT-SEN UNIVERSITY

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

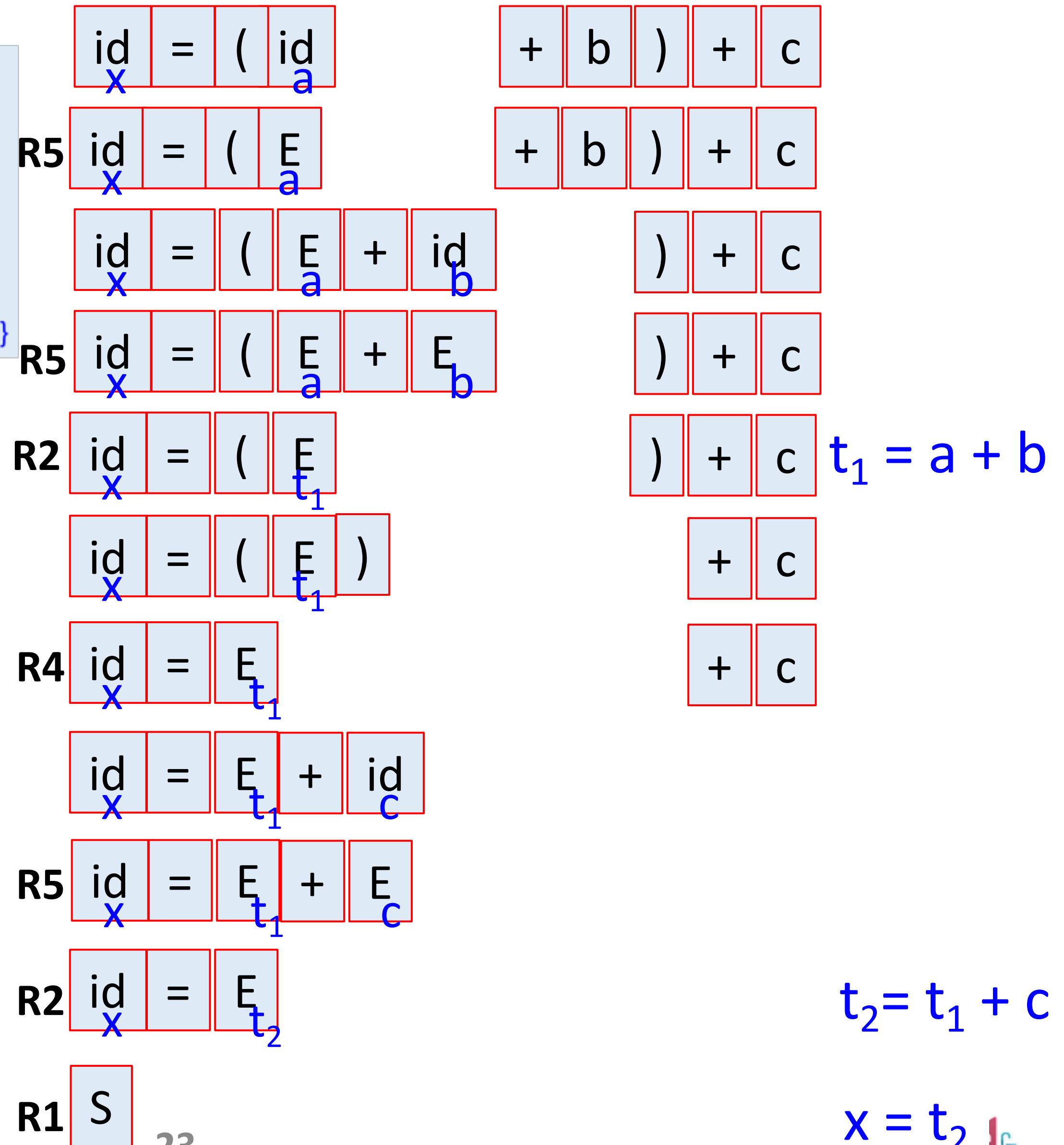
  $x = (a + b) + c$

- Translated TAC

  $t_1 = a + b$

  $t_2 = t_1 + c$

  $x = t_2$

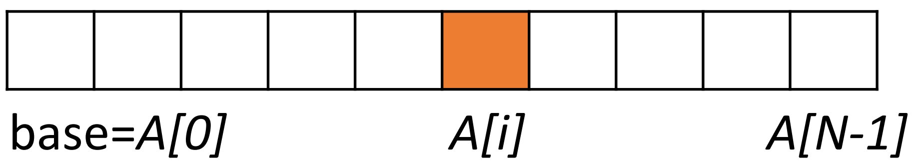


CodeGen: Array Reference[数组引用]

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

  int A[N];

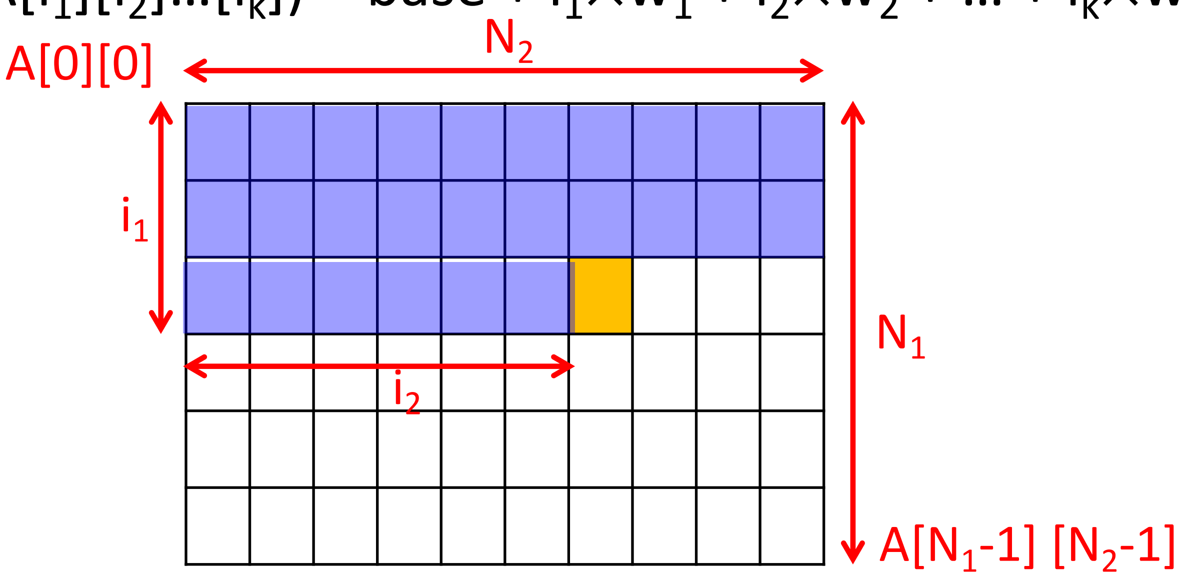
  A[i] ++;



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

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))
  - $c = a[i_1][i_2];$

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

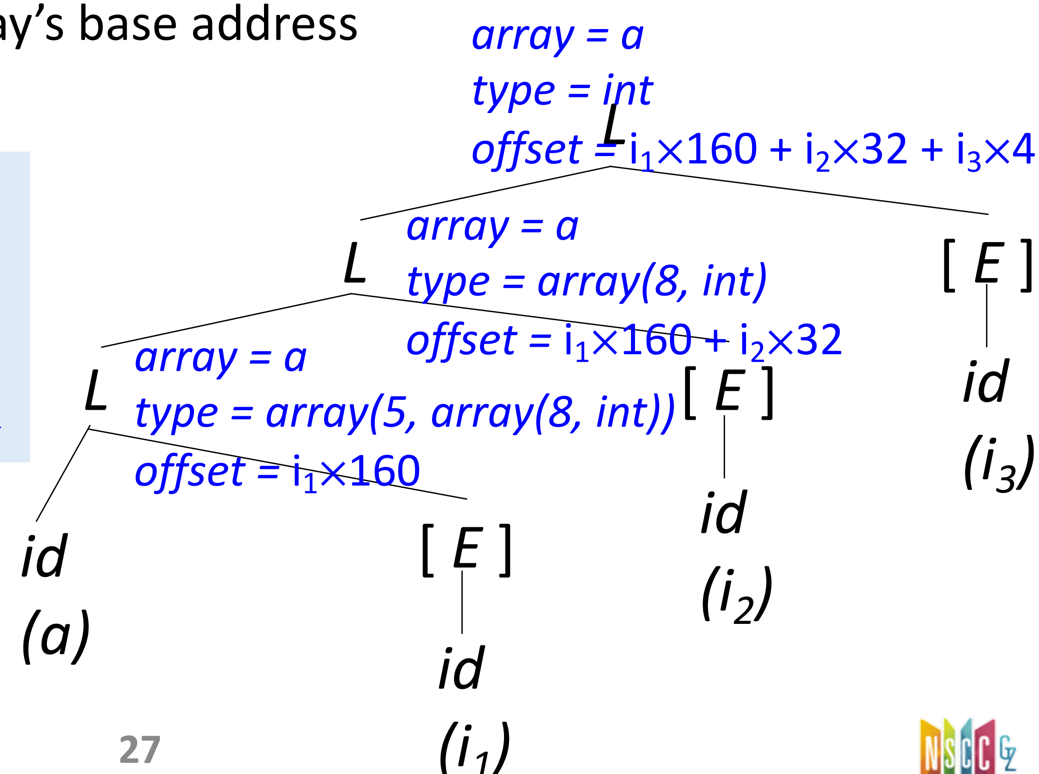
- Type(a) = array(3, array(5, array(8, int)))
  - $c = a[i_1][i_2][i_3]$

$$
\begin{aligned}
addr(a[i_1][i_2][i_3]) &= base + i_1*w_1 + i_2*w_2 + i_3*w_3 \\
&= base + i_1*160 + i_2*32 + i_3*4
\end{aligned}
$$

Translation of Array References (cont.)

- $A[i_1][i_2][i_3]$, type(a) = array(3, 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*]
$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$



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 -> E_1 + E_2 | - E_1 | (E_1) | 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); }
| L_1 [E] { L.array = L_1.array;
L.type = L_1.type.elem;
t = newtemp();
gen(t '=' E.addr '*' L.type.width);
L.addr = newtemp();
gen(L.addr '=' L_1.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)

t_1 = a < b
t_2 = c < d
t_3 = e < f
t_4 = t_2 && t_3
t_5 = t_1 || t_4

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

S -> if E S_1

if (!t_5) goto *S.next*
S_1.code
S.next: ...

Boolean Exprs (w/ Short-Circuiting)

- Implemented via a series of jumps[利用跳转]
  - Each relational op converted to two gotos (*true* and *false*)
  - Remaining evaluation skipped when result known in middle
- Example
  - *E.true*: label for code to execute when *E* is *'true'*
  - *E.false*: label for code to execute when *E* is *'false'*
  - E.g. if above is condition for a *while* loop
    - *E.true* would be label at beginning of loop body
    - *E.false* would be label for code after the loop

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

if (a < b) goto *E.true*	E为真：只要a < b真
goto L_1	a < b假：继续评估
L_1: if (c < d) goto L_2	a < b假、 c < d真：继续评估
goto *E.false*	E为假：a < b假，c < d假
L_2: if (e < f) goto *E.true*	E为真：a < b假，c < d真，e < f真
goto *E.false*	E为假：a < b假，c < d真，e < f假