中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第20讲：中间代码(2)

张献伟

xianweiz.github.io

DCS290, 5/9/2023

Review Questions

- Input and output of code generation?

  Input: AST + symbol table; output: IR

- What is IR?

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

- Why do we use IR?

  Clean separation of front-/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?

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

Three-Address Code[三地址码]

- High-level assembly where each operation has **at most three** operands. Generic form is X = Y op Z[最多3个操作数]
  - where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values
- Characteristics[特性]
  - Assembly code for an 'abstract machine'
  - Long expressions are converted to multiple instructions
  - Control flow statements are converted to jumps[控制流->跳转]
  - Machine independent
    - Operations are generic (not tailored to any specific machine)
    - Function calls represented as generic call nodes
    - Uses symbolic names rather than register names (actual locations of symbols are yet to be determined)
- Design goal: for easier machine-independent optimization

Three-Address Statements

- Assignment statement[二元赋值]

  x = y op z

  where op is an arithmetic or logical operation (binary operation)

- Assignment statement[一元赋值]

  x = op y

  where op is an unary operation such as -, not, shift

- Copy statement[拷贝]

  x = y

- Unconditional jump statement[无条件跳转]

  goto L

  where L is label

Three-Address Statements (cont.)

- Conditional jump statement[条件跳转]

  if (x relop y) goto L

  where relop is a relational operator such as =, ≠, >, <

- Procedural call statement[过程调用]: may have too many addr

  param x_1, ..., param x_n, call F_y, n

  As an example, foo(x_1, x_2, x_3) is translated to

  param x_1

  param x_2

  param x_3

  call foo, 3

- Procedural call return statement[过程调用返回]

  return y

  where y is the return value (if applicable)

Three-Address Statements (cont.)

- Indexed assignment statement[索引]

  x = y[i]

  or

  y[i] = x

  where x is a scalar variable and y is an array variable

- Address and pointer operation statement[地址和指针]

  x = & y ; a pointer x is set to address of y

  y = * x ; y is set to the value of location

  ; pointed to by pointer x

  *y = x ; location pointed to by y is assigned x

Example: TAC

```
i = 1
do {
   a[i] = x * 5;
   i ++;
} while (i <= 10);
```

Source program

$i = 1$

L: $t_1 = x * 5$

$t_2 = \&a$

$t_3 = sizeof(int)$

$t_4 = t_3 * i$

$t_5 = t_2 + t_4$

$*t_5 = t_1$

$i = i + 1$

if $i <= 10$ goto L

a[i] (covers t_2 through t_5)

Three-address code

Example: TAC (cont.)

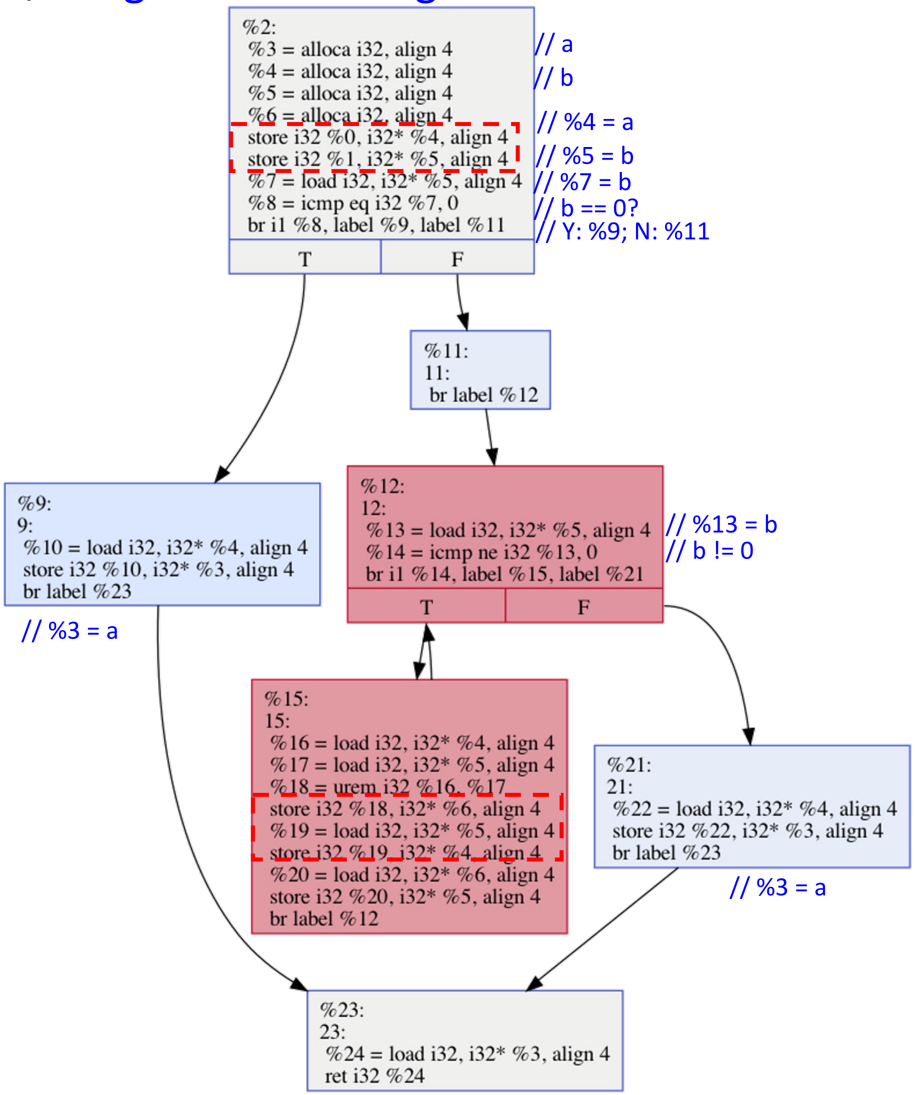
```
i = 1
do {
  a[i] = x * 5;
  i ++;
} while (i <= 10);
```

```c
int i = 1, x = 2;
int a[10];

int main(){

  do {
    a[1] = x * 5;
    i++;
  } while (i <= 10);

  return 0;
}
```

```
@i = dso_local global i32 1, align 4
@x = dso_local global i32 2, align 4
@a = dso_local global [10 x i32] zeroinitializer, align 4

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  br label %2

2:                                                ; preds = %7, %0
  %3 = load i32, i32* @x, align 4      // %3 = x
  %4 = mul nsw i32 %3, 5               // %4 = %3 x 5
  store i32 %4, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @a, i64 0, i64 1), align 4
                                       // addr(@a + 0 + 1*4) = %4
  %5 = load i32, i32* @i, align 4      // %5 = i
  %6 = add nsw i32 %5, 1               // %6 = %5 + 1
  store i32 %6, i32* @i, align 4       // i = %6
  br label %7

7:                                                ; preds = %2
  %8 = load i32, i32* @i, align 4      // %8 = i
  %9 = icmp sle i32 %8, 10             // %9 = (i <= 10)
  br i1 %9, label %2, label %10        // T: %2, F: %10

10:                                               ; preds = %7
  ret i32 0
}
```

Example: IR and SSA

$clang -emit-llvm -S gcd.c

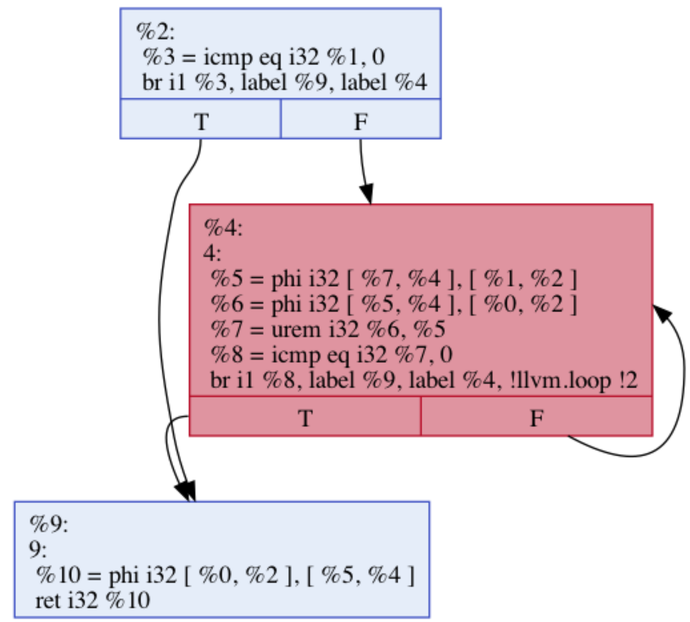


CFG for 'gcd' function

Load-and-store approach (not SSA)

```c
unsigned gcd(unsigned a, unsigned b) {
  if (b == 0)
    return a;
  while (b != 0) {
    unsigned t = a % b;
    a = b;
    b = t;
  }
  return a;
}
```

$clang -emit-llvm -S -O1 gcd.c

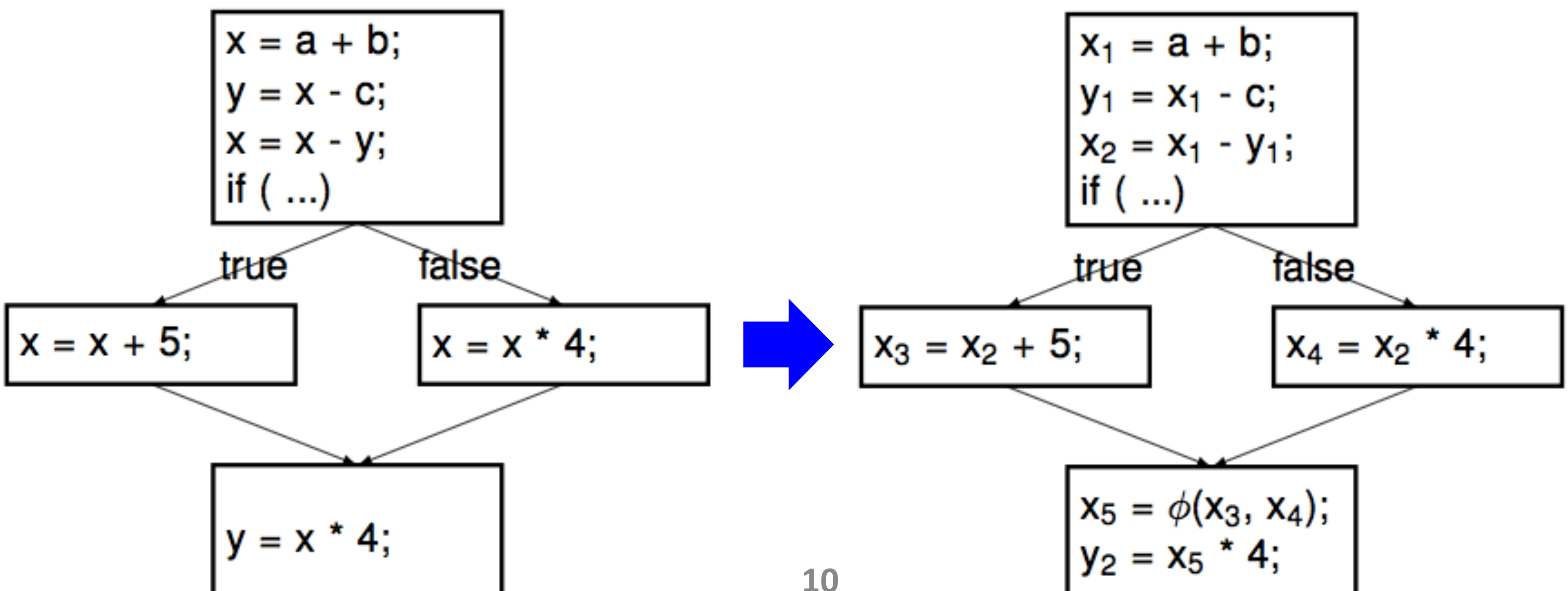


CFG for 'gcd' function

Phi approach (SSA)

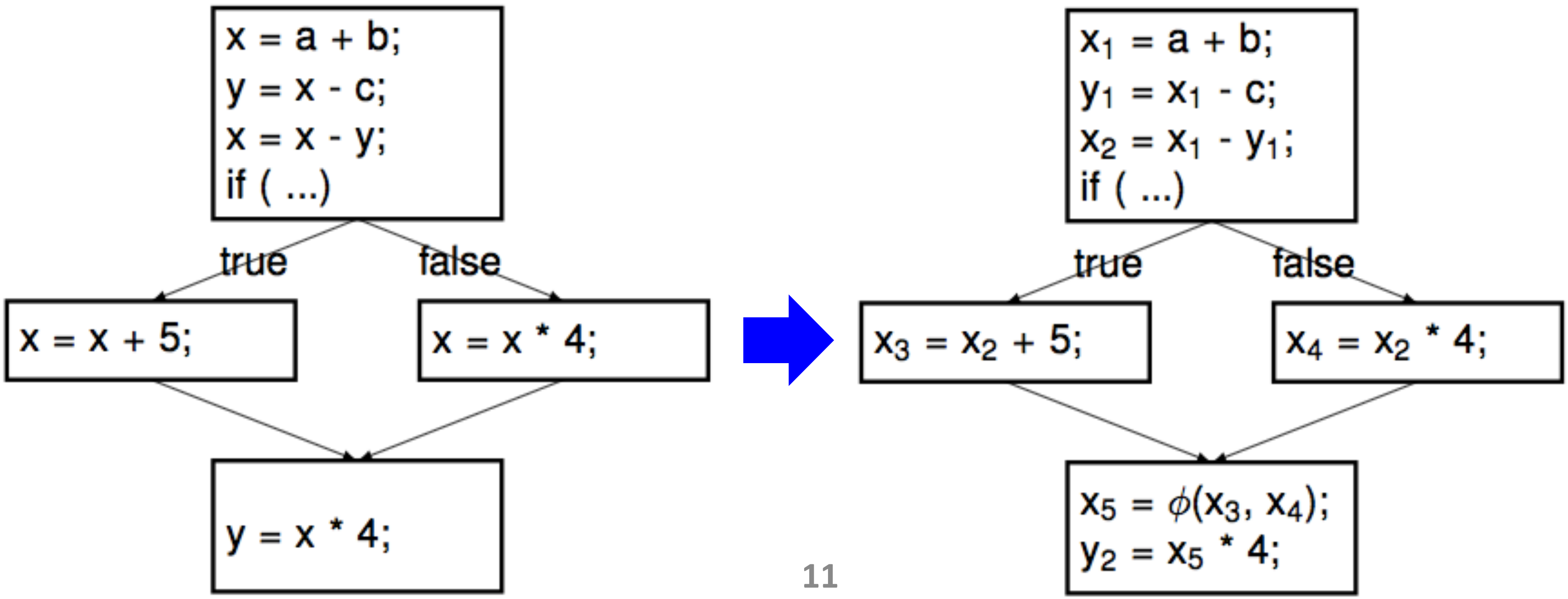
Single Static Assignment[静态单赋值]

- Every variable is assigned to exactly once statically[仅一次]
  - Give variable a different version name on every assignment
    - e.g. $x \rightarrow x_1, x_2, ..., x_5$ 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)

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[数据流显现]
  - Without SSA, need a separate dataflow graph
  - Will discuss more in **Compiler Optimization** section

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;
```

→

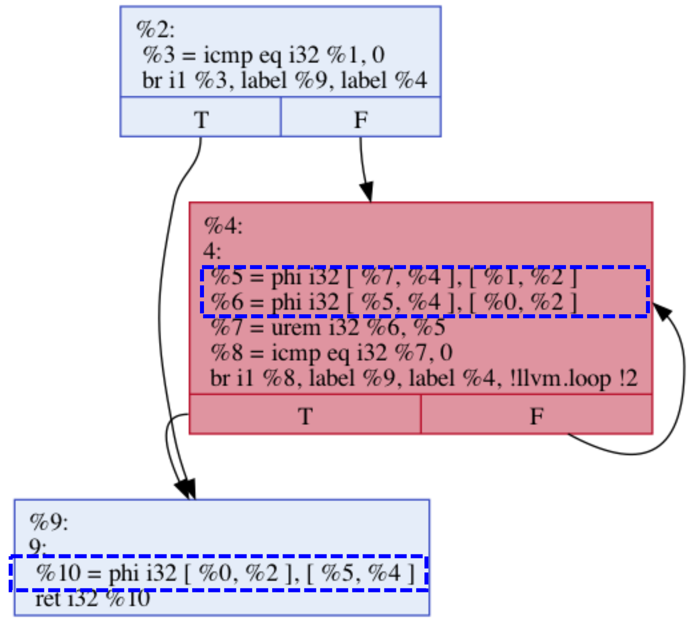
a_1 = 1;
if (v < 10)
   a_2 = 2;
b = PHI(a_1, a_2);

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

Example

- Registers
  - Unlimited #virtual registers
  - Each is written only once
  - %0: *a*, %1: *b*
- Phi instructions
  - %5 = phi i32 [%7, %4], [%1, %2]
    - *b* is from before-while or while
  - %6 = phi i32 [%5, %4], [%0, %2]
    - *a* is either before-while or while
  - %10 = phi i32 [%0, %2], [%5, %4]
    - *a* is either before-while or while
- Phi restrictions
  - Must be the 1st insts of a BB
  - The 1st BB cannot begin with phi
    - Has no previously executed block

$clang -emit-llvm -S -O1 gcd.c



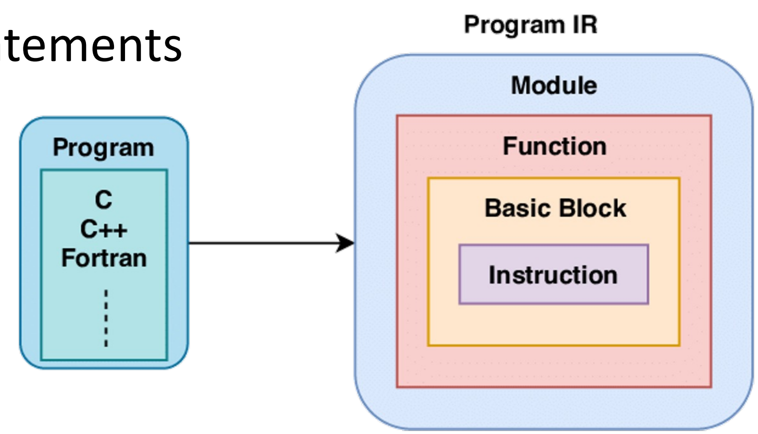
CFG for 'gcd' function

Phi approach (SSA)

```c
unsigned gcd(unsigned a, unsigned b) {
  if (b == 0)
    return a;
  while (b != 0) {
    unsigned t = a % b;
    a = b;
    b = t;
  }
  return a;
}
```

IR 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...?

Example: LLVM

```c
double x;

void foo() {
    char a;
    int b = 0;
    long long c;
    int d;
}
```

```llvm
@x = dso_local global double 0.000000e+00, align 8

; Function Attrs: noinline nounwind optnone
define dso_local void @foo() #0 {
  %1 = alloca i8, align 1
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  ret void
}
```

```cpp
auto addr = Builder.CreateAlloca(…);
Builder.CreateStore(…, addr);
```

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

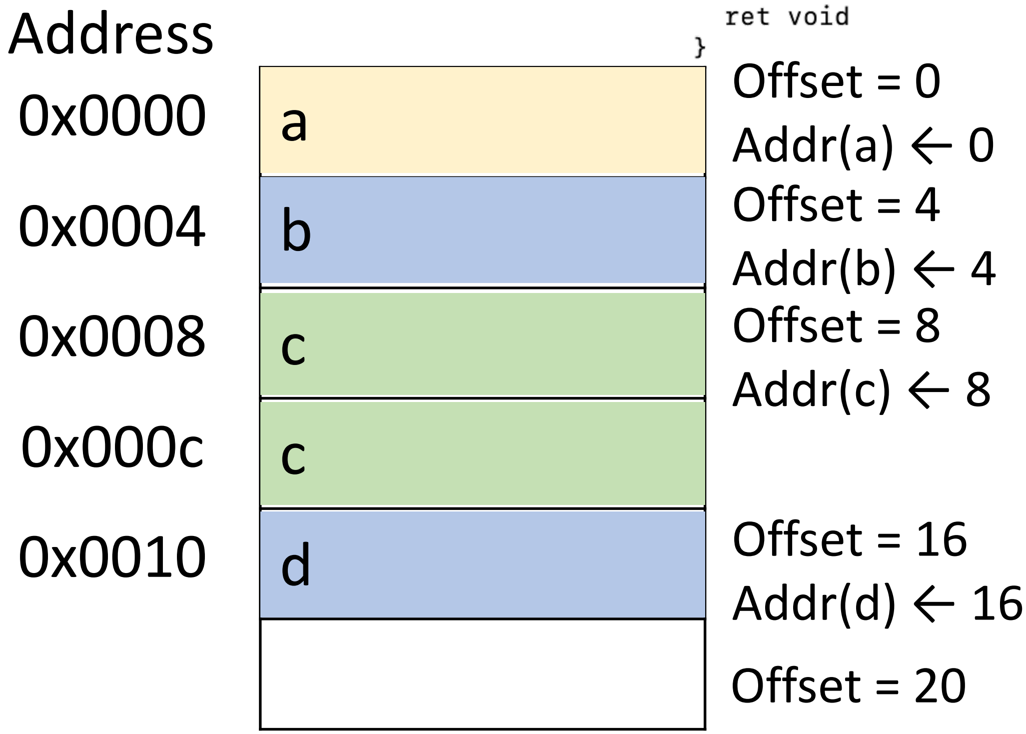
- **Solution**: allocate memory briefly for each scope[域内]
  - Allocate when entering scope, free when exiting scope
  - Variables in the same scope are allocated / freed together
  - Location = var offset + base of scope memory section
  - Will discuss more later in **Runtime Management**

Storage Layout of Variables in a Function

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

```
define dso_local void @foo() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  ret void
}
```

```
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
}
```

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

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;* }

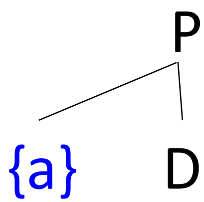
⑦ B -> real { *B.type = real; B.width = 8;* }

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;* }

Example

- Input: real x; int i;

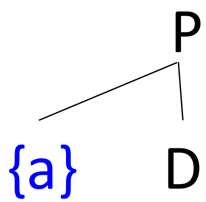


① P -> { $offset = 0$ } D
② D -> T id; { $enter(id.lexeme, T.type, offset);$
$offset = offset + T.width;$ } D_1
③ D -> ε
④ T -> B { $t = B.type; w = B.width;$ }
C { $T.type = C.type; T.width = C.width;$ }
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type); T.width = 4;$ }
⑥ B -> int { $B.type = int; B.width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

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; }

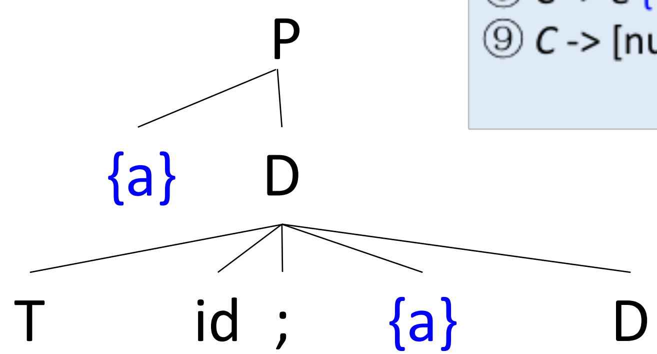


offset = 0

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;* }

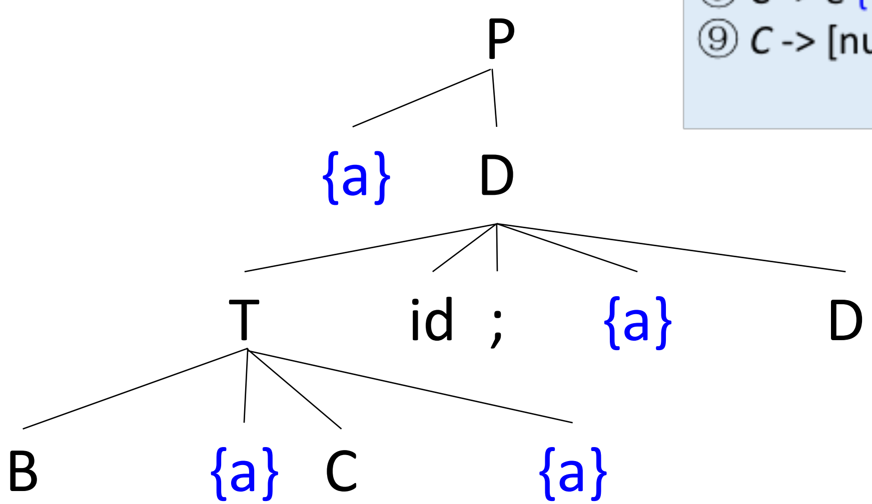


offset = 0

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

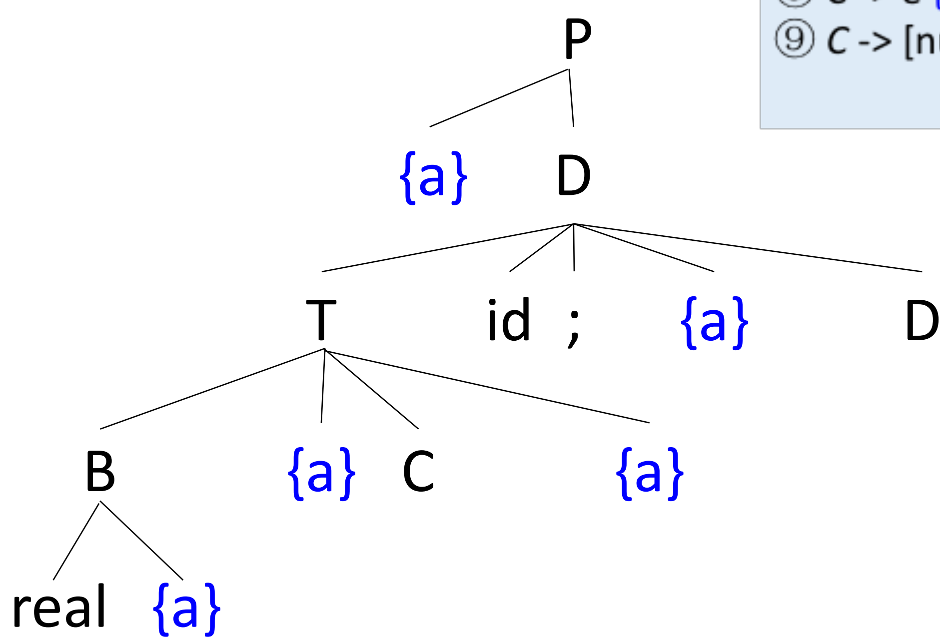


offset = 0

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

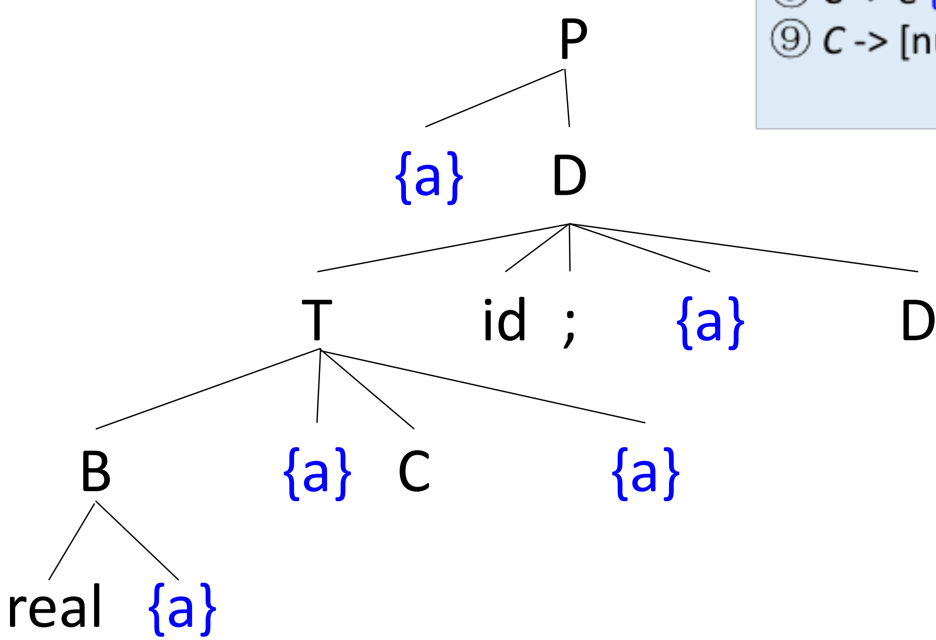


offset = 0

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; }

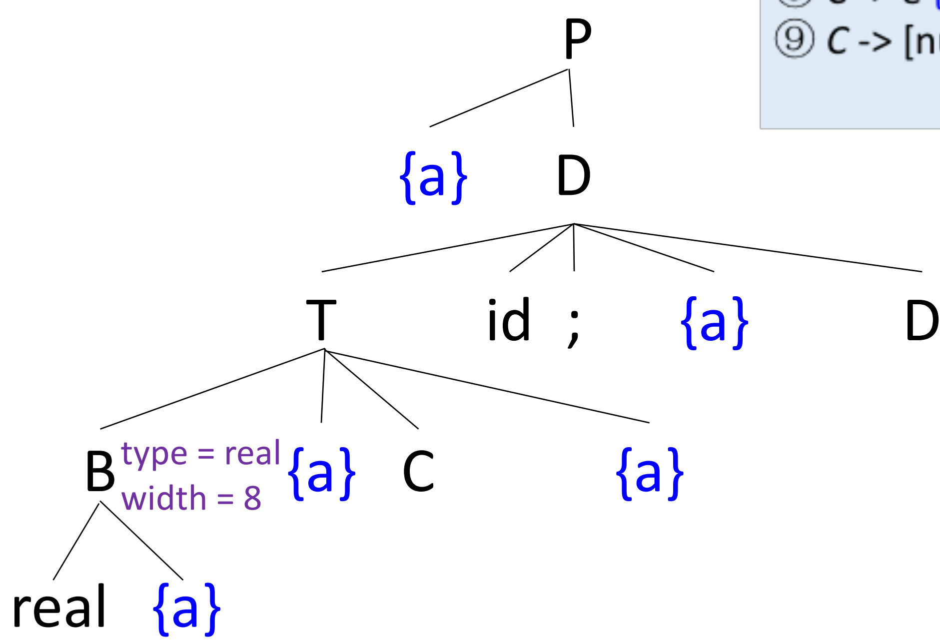


offset = 0

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



offset = 0

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 { $T.type = C.type; T.width = C.width;$ }

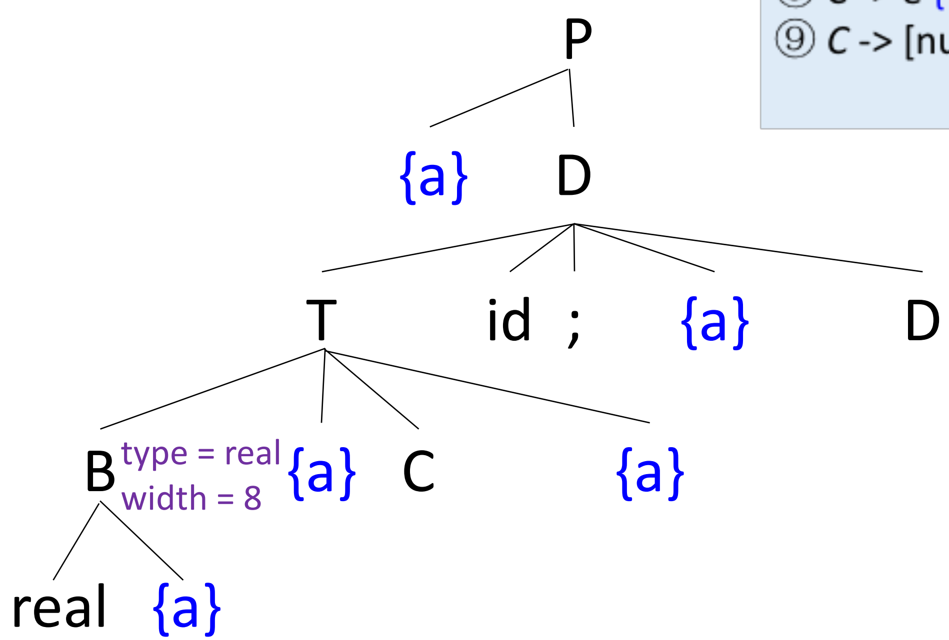
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type); T.width = 4;$ }

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



offset = 0

t = real
w = 8

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;* }

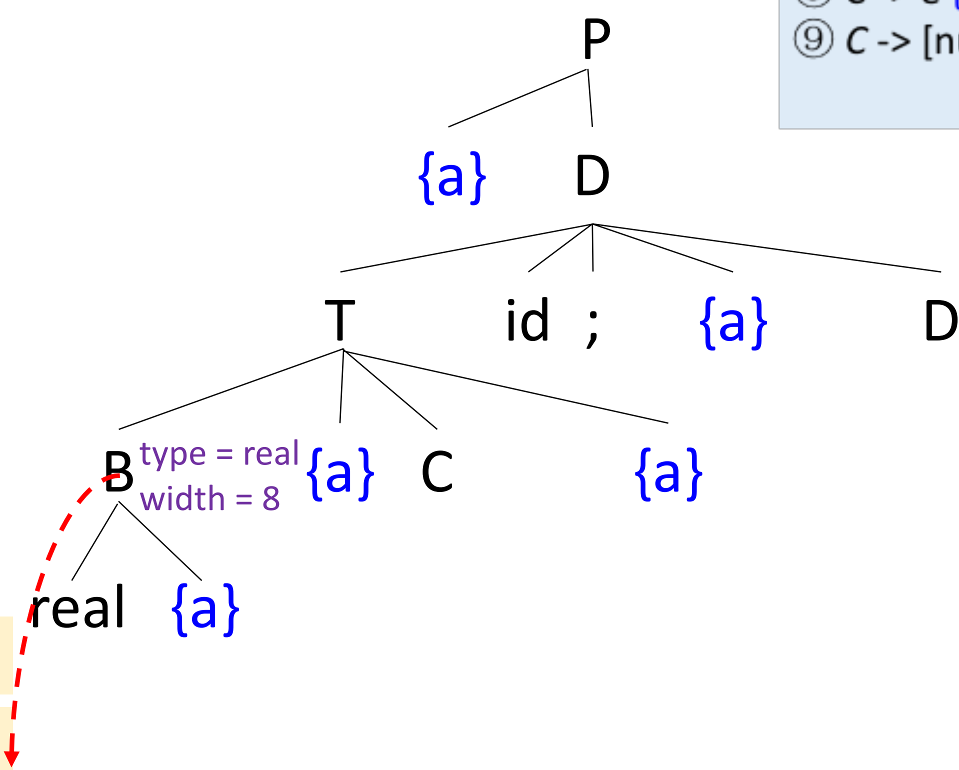
⑤ T -> *T_1 { *T.type = pointer(* T_1*.type); T.width = 4;* }

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;* }



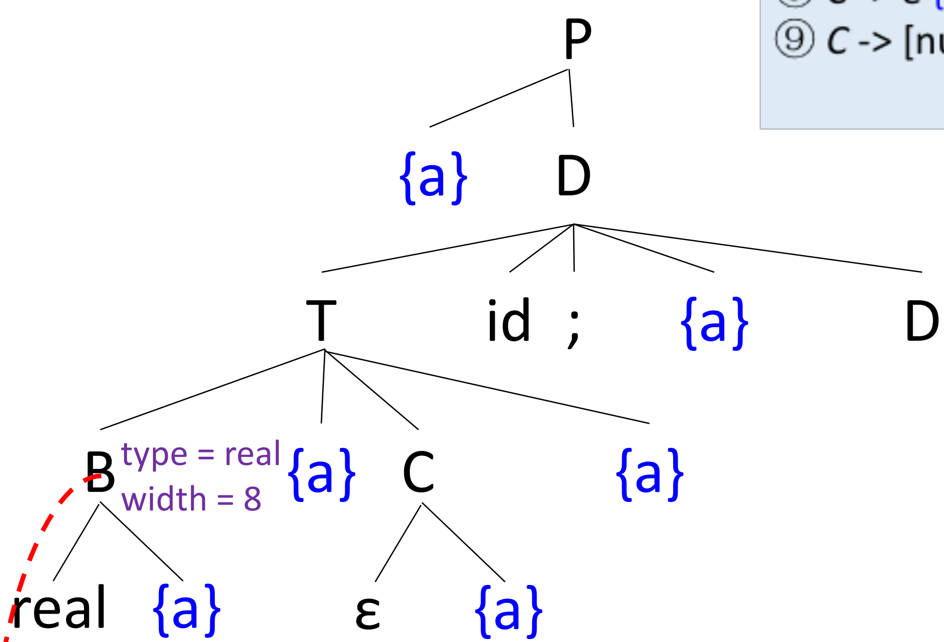
offset = 0

t = real
w = 8

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



$offset = 0$

$t = real$
$w = 8$

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 { $T.type = C.type; T.width = C.width;$ }

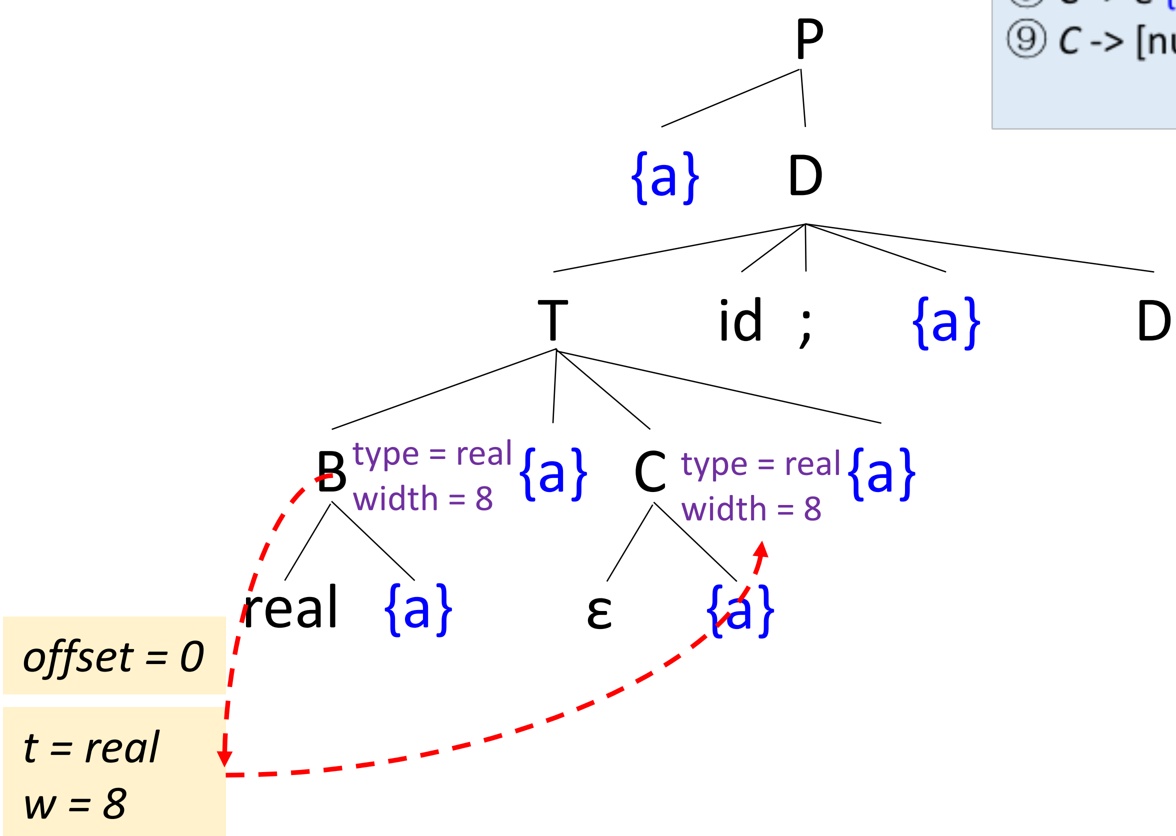
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type); T.width = 4;$ }

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



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 { $T.type = C.type; T.width = C.width;$ }

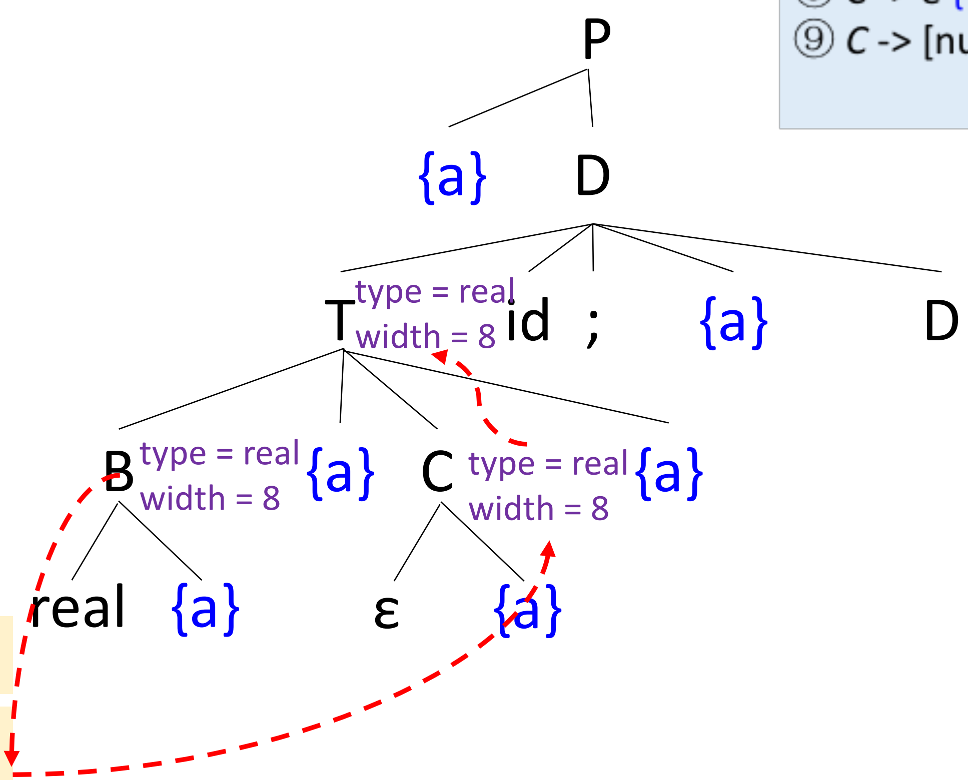
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type); T.width = 4;$ }

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



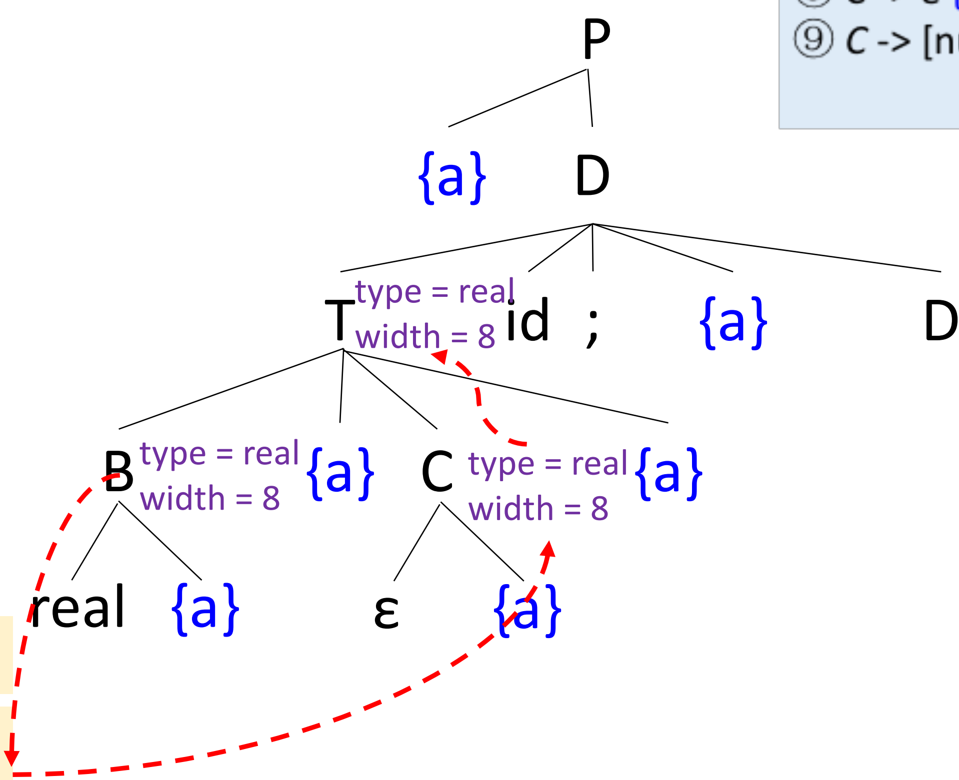
offset = 0

t = real
w = 8

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; }



offset = 0

t = real
w = 8

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; }

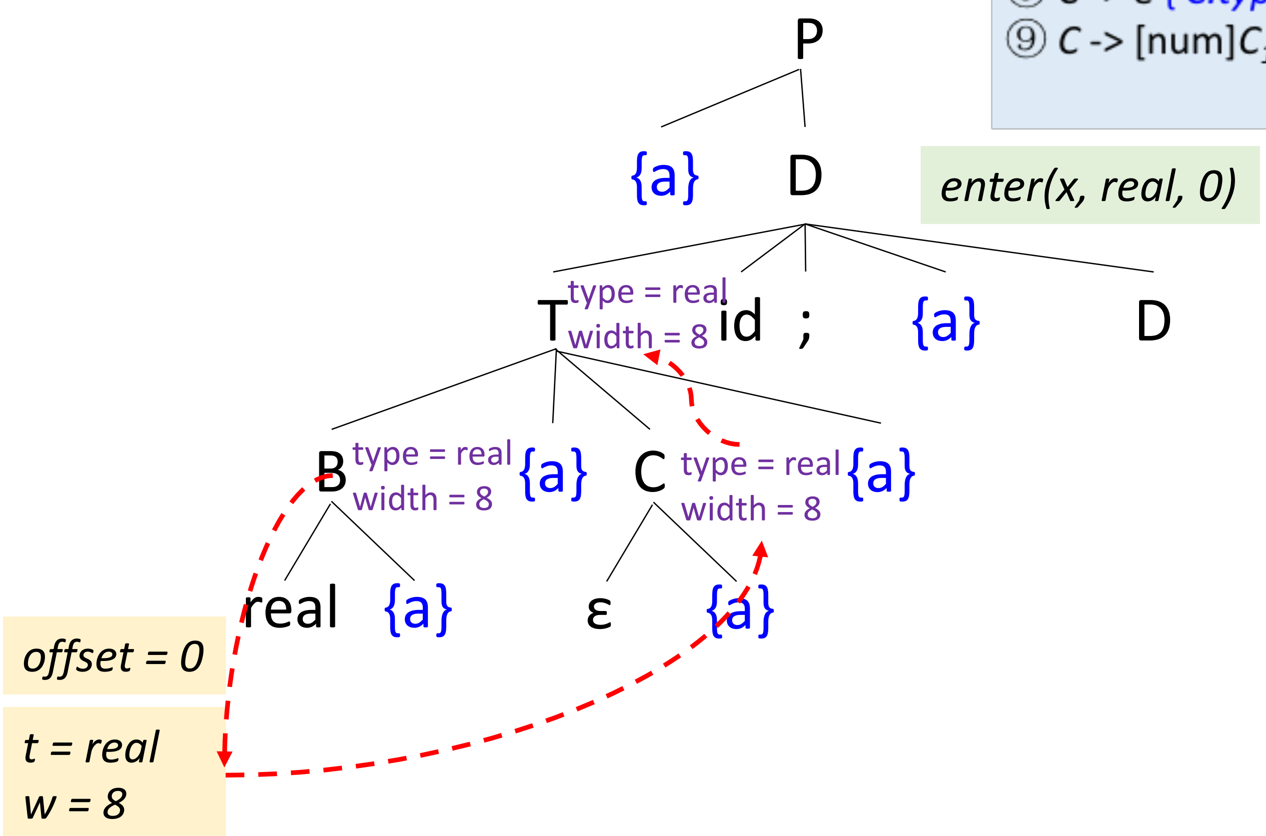
⑤ T -> *T_1 { T.type = pointer(T_1.type); T.width = 4; }

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; }



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 { $T.type = C.type; T.width = C.width;$ }

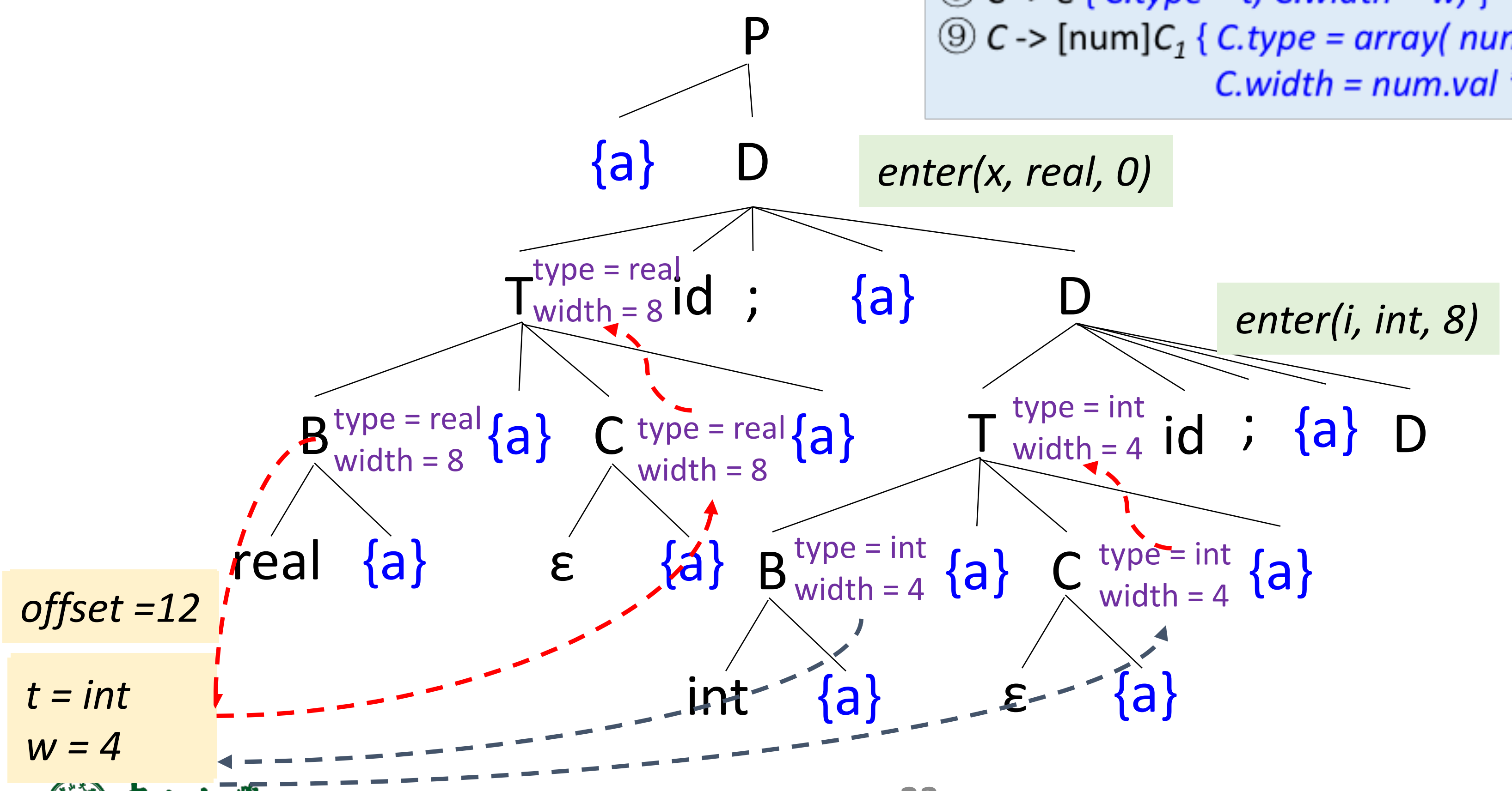
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type); T.width = 4;$ }

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]

- Translations
  - 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
- We can also use the syntax-directed formalisms to specify translations

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

Example: LLVM

```c
double x;

void foo() {
    char a;
    int b = 0;
    long long c;
    int d;

    int x = b + (-d);
}
```

```llvm
@x = dso_local global double 0.000000e+00, align 8

; Function Attrs: noinline nounwind optnone
define dso_local void @foo() #0 {
  %1 = alloca i8, align 1
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4          // int x
  store i32 0, i32* %2, align 4
  %6 = load i32, i32* %2, align 4   // %6 = b
  %7 = load i32, i32* %4, align 4   // %7 = d
  %8 = sub nsw i32 0, %7            // %8 = -d
  %9 = add nsw i32 %6, %8           // %9 = b + (-d)
  store i32 %9, i32* %5, align 4    // x = %9 = b + (-d)
  ret void
}
```

```cpp
auto left = myBuildExp(…);
auto right = myBuildExp(…);
Builder.CreateAdd(left, right, “add”);
```

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;*
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;*
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 = ''; }

Code attributes can be long strings

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

Code attributes can be long strings

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;*
}

Example

id (x)	=	(	id (a)	+	b	)	+	c

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

Example

	id (x)	=	(	id (a)		+	b	)	+	c
R5	id (x)	=	(	E (a)		+	b	)	+	c

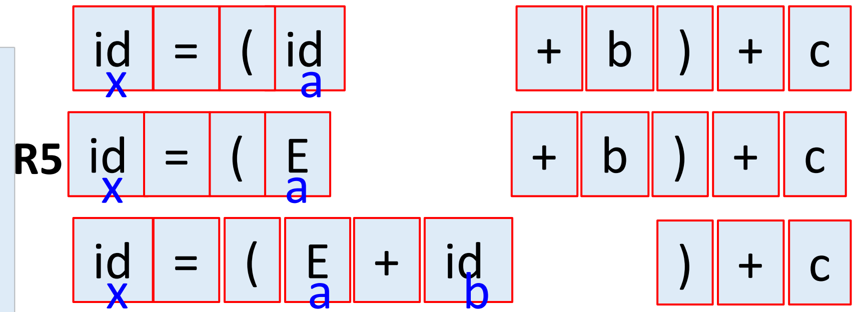
① S -> id = E; { *p = lookup(*id*.lexeme); if !p then error;*
gen(p '=' E.addr); }
② E -> E_1 + E_2; { *E.addr = newtemp();*
gen(E.addr '=' E_1.addr '+' E_2.addr); }
③ E -> - E_1 { *E.addr = newtemp();*
gen(E.addr '=' 'minus' E_1.addr); }
④ E -> (E_1) { *E.addr = E_1.addr; }*
⑤ E -> id { *E.addr = lookup(*id*.lexeme);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

- Input

  x = (a + b) + c

Example

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

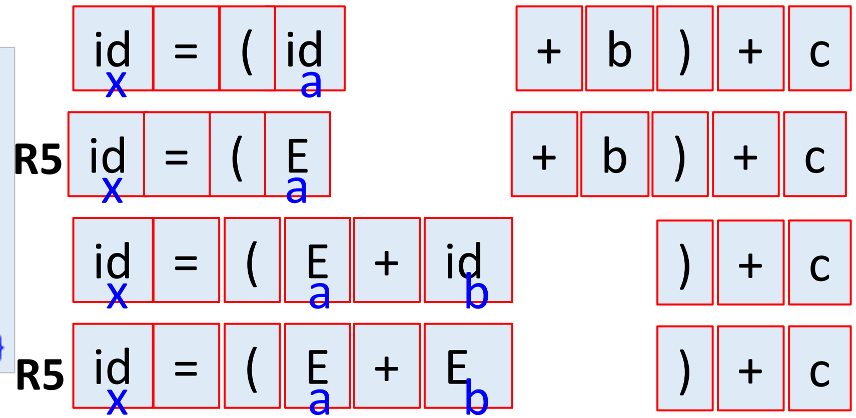


- Input

  x = (a + b) + c

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; }

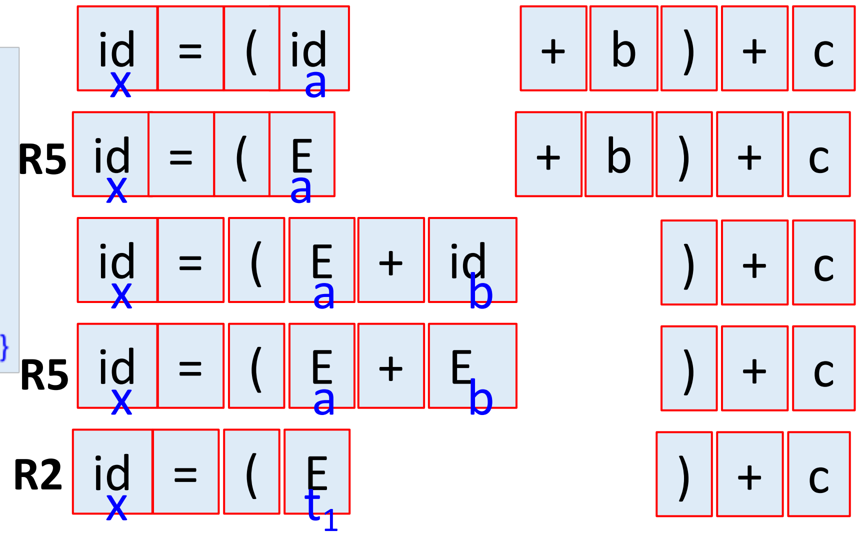


- Input

x = (a + b) + c

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;* }

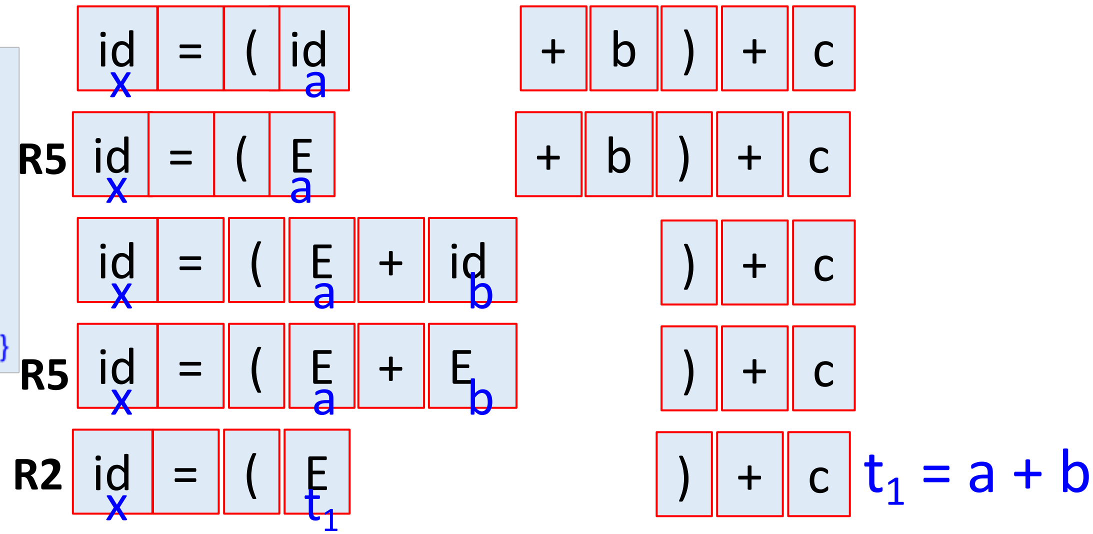


- Input

  x = (a + b) + c

Example

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

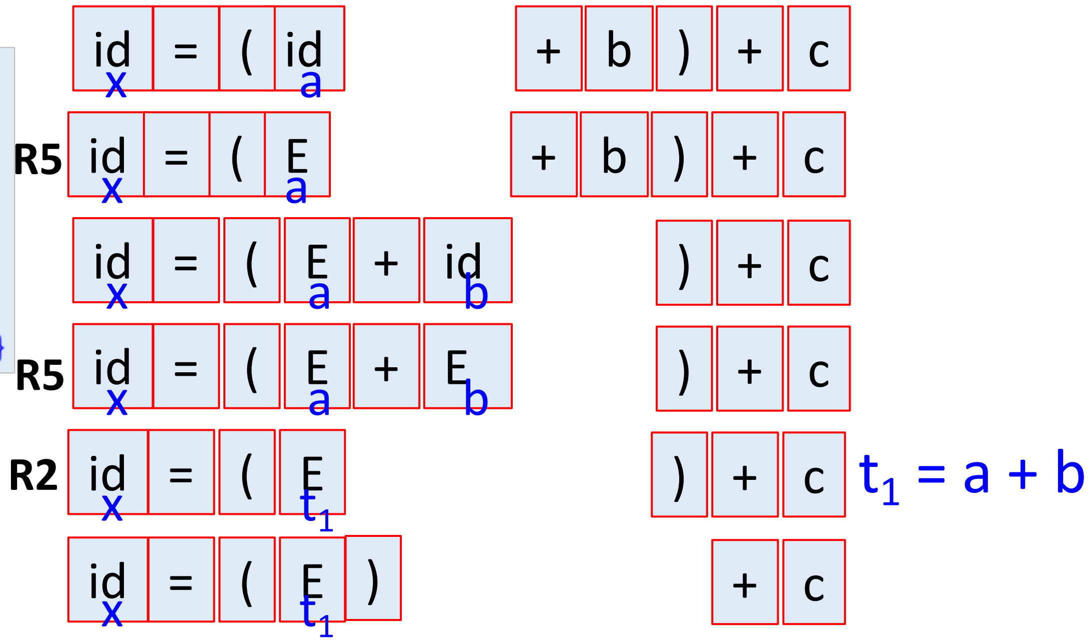


- Input

$x = (a + b) + c$

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

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

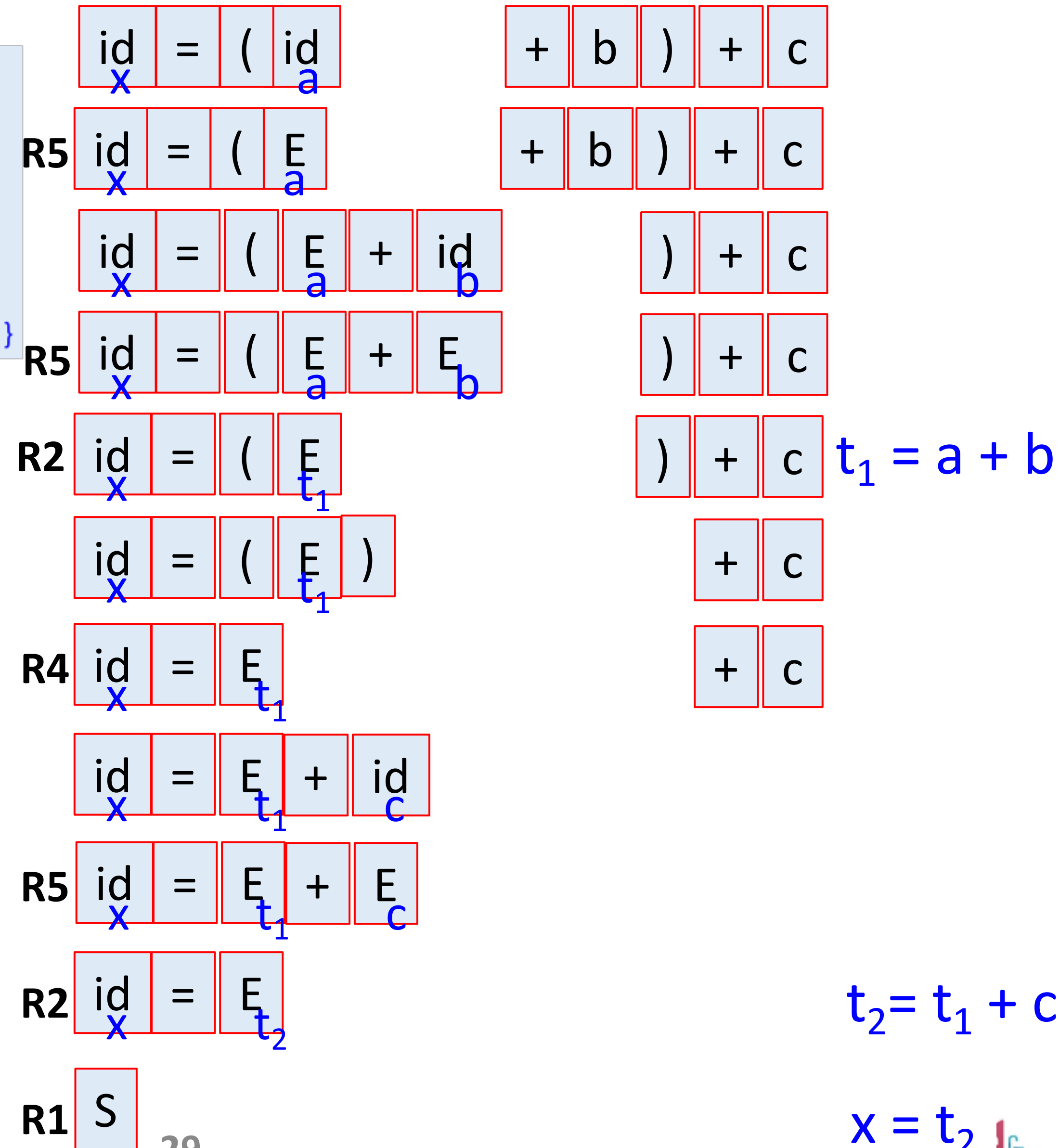
  x = (a + b) + c

- Translated TAC

  $t_1 = a + b$

  $t_2 = t_1 + c$

  $x = t_2$

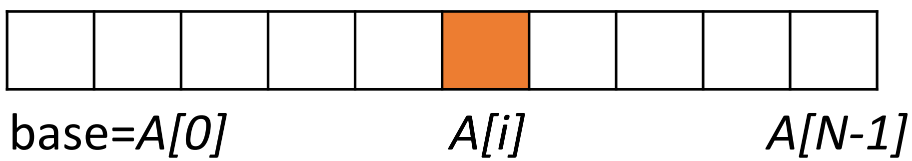


CodeGen: Array Reference[数组引用]

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

  int A[N];

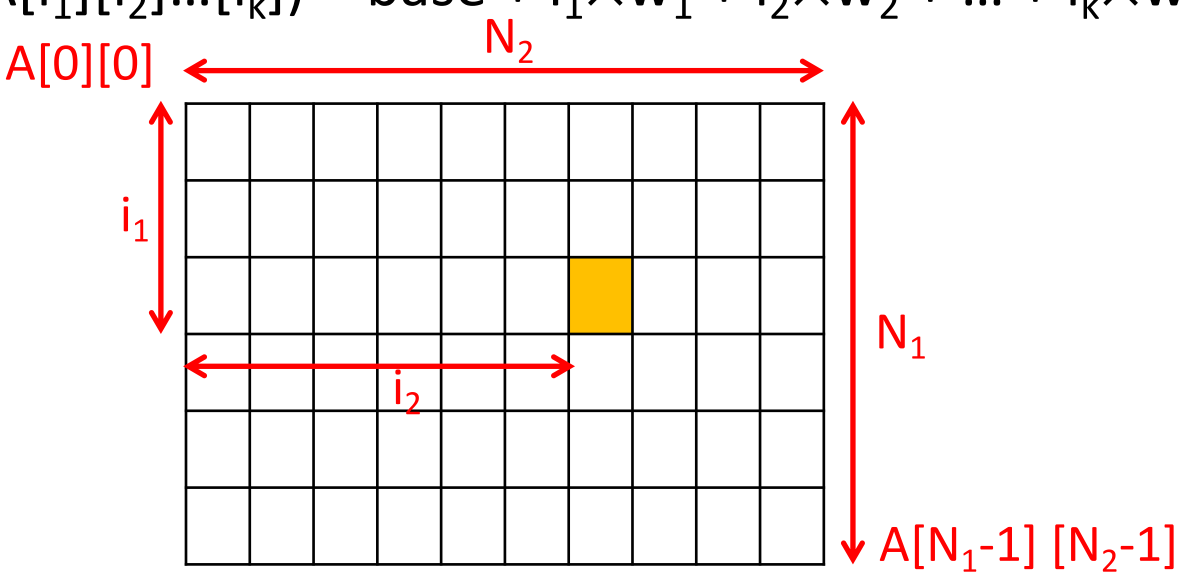
  A[i] ++;



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

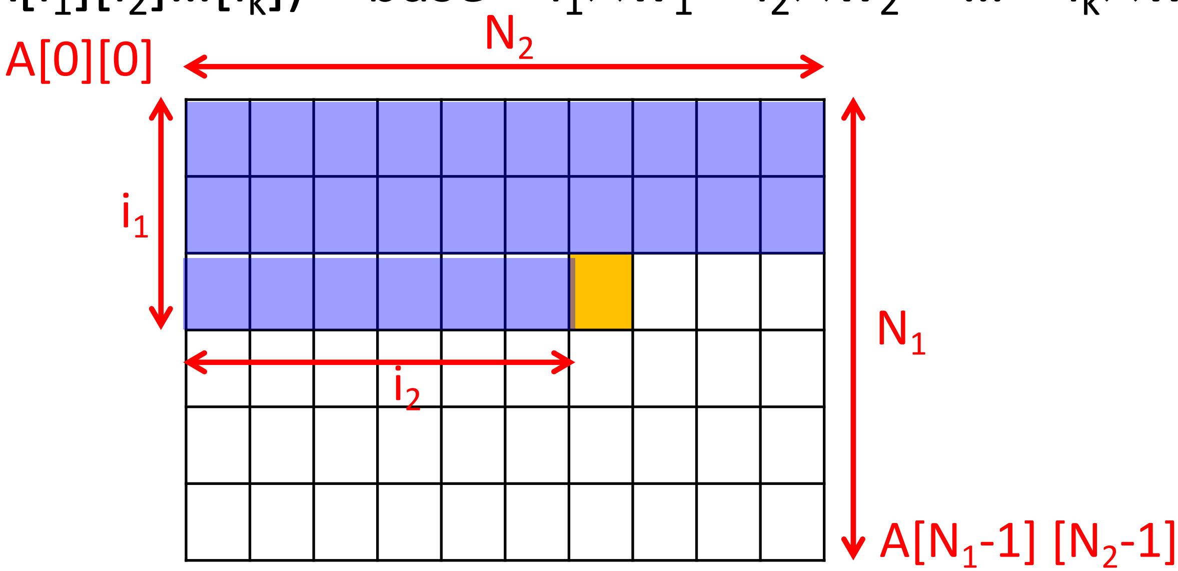
N-dimensional Array

- 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 <u>row-major</u> or <u>column-major</u>
  - C language uses row major (i.e., stored row by row)
  - Row-major: addr($A[i_1, i_2]$) = base + ($i_1 \times \underbrace{N_2 * width}_{w_1} + i_2 \times \underbrace{width}_{w_2}$)
- *k*-dimensional array
  - addr($A[i_1][i_2]...[i_k]$) = base + $i_1 \times w_1 + i_2 \times w_2 + ... + i_k \times w_k$



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 <u>row-major</u> or <u>column-major</u>
  - C language uses row major (i.e., stored row by row)
  - Row-major: addr($A[i_1, i_2]$) = base + ($i_1 \times \underbrace{N_2 * width}_{w_1} + i_2 \times \underbrace{width}_{w_2}$)
- *k*-dimensional array
  - addr($A[i_1][i_2]...[i_k]$) = base + $i_1 \times w_1 + i_2 \times w_2 + ... + i_k \times w_k$



Example: LLVM

```c
double x;
int arr[3][5][8];

void foo() {
    char a;
    int b = 0;
    long long c;
    int d;

    int x = arr[2][3][4];
}
```

```llvm
@arr = dso_local global [3 x [5 x [8 x i32]]] zeroinitializer, align 4
@x = dso_local global double 0.000000e+00, align 8

; Function Attrs: noinline nounwind optnone
define dso_local void @foo() #0 {
  %1 = alloca i8, align 1
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %2, align 4                  // addr(@arr + 4x(0 + 2*3*4 + 3*4 + 4) )
  %6 = load i32, i32* getelementptr inbounds ([3 x [5 x [8 x i32]]], [3
x [5 x [8 x i32]]]* @arr, i64 0, i64 2, i64 3, i64 4), align 4
  store i32 %6, i32* %5, align 4
  ret void
}
```

Builder.CreateInBoundsGEP(addr, ...);