Compilation Principle
编 译 原 理

第19讲：中间代码(2)

张献伟

xianweiz.github.io

DCS290, 5/16/2024

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)?

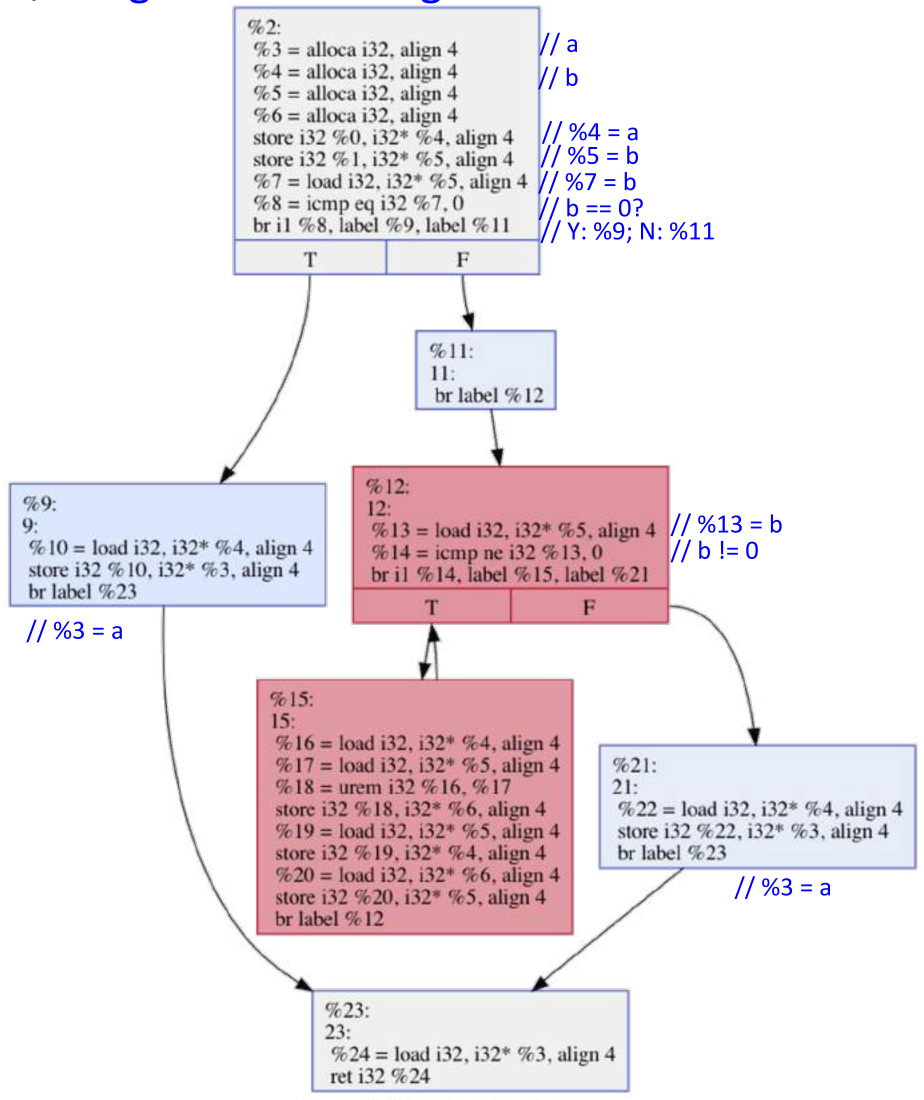
  A type of IR, with at most three operands. (high-level assembly)

- TAC of x + y * z + 5?

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

Example: IR and SSA

$clang -emit-llvm -S gcd.c

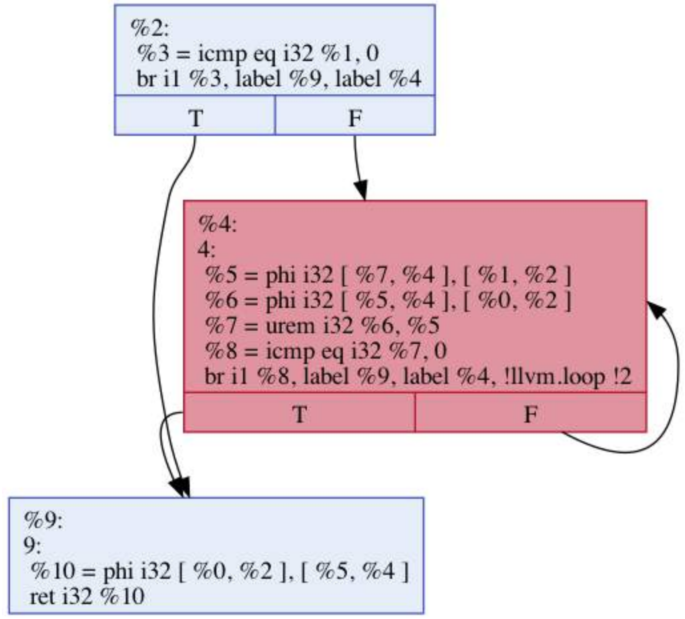


```
%2:
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i32, align 4
 %6 = alloca i32, align 4
 store i32 %0, i32* %4, align 4
 store i32 %1, i32* %5, align 4
 %7 = load i32, i32* %5, align 4
 %8 = icmp eq i32 %7, 0
 br i1 %8, label %9, label %11

%11:
11:
 br label %12

%9:
9:
 %10 = load i32, i32* %4, align 4
 store i32 %10, i32* %3, align 4
 br label %23

%12:
12:
 %13 = load i32, i32* %5, align 4
 %14 = icmp ne i32 %13, 0
 br i1 %14, label %15, label %21

%15:
15:
 %16 = load i32, i32* %4, align 4
 %17 = load i32, i32* %5, align 4
 %18 = urem i32 %16, %17
 store i32 %18, i32* %6, align 4
 %19 = load i32, i32* %5, align 4
 store i32 %19, i32* %4, align 4
 %20 = load i32, i32* %6, align 4
 store i32 %20, i32* %5, align 4
 br label %12

%21:
21:
 %22 = load i32, i32* %4, align 4
 store i32 %22, i32* %3, align 4
 br label %23

%23:
23:
 %24 = load i32, i32* %3, align 4
 ret i32 %24
```

CFG for 'gcd' function

```
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



```
%2:
 %3 = icmp eq i32 %1, 0
 br i1 %3, label %9, label %4

%4:
4:
 %5 = phi i32 [ %7, %4 ], [ %1, %2 ]
 %6 = phi i32 [ %5, %4 ], [ %0, %2 ]
 %7 = urem i32 %6, %5
 %8 = icmp eq i32 %7, 0
 br i1 %8, label %9, label %4, !llvm.loop !2

%9:
9:
 %10 = phi i32 [ %0, %2 ], [ %5, %4 ]
 ret i32 %10
```

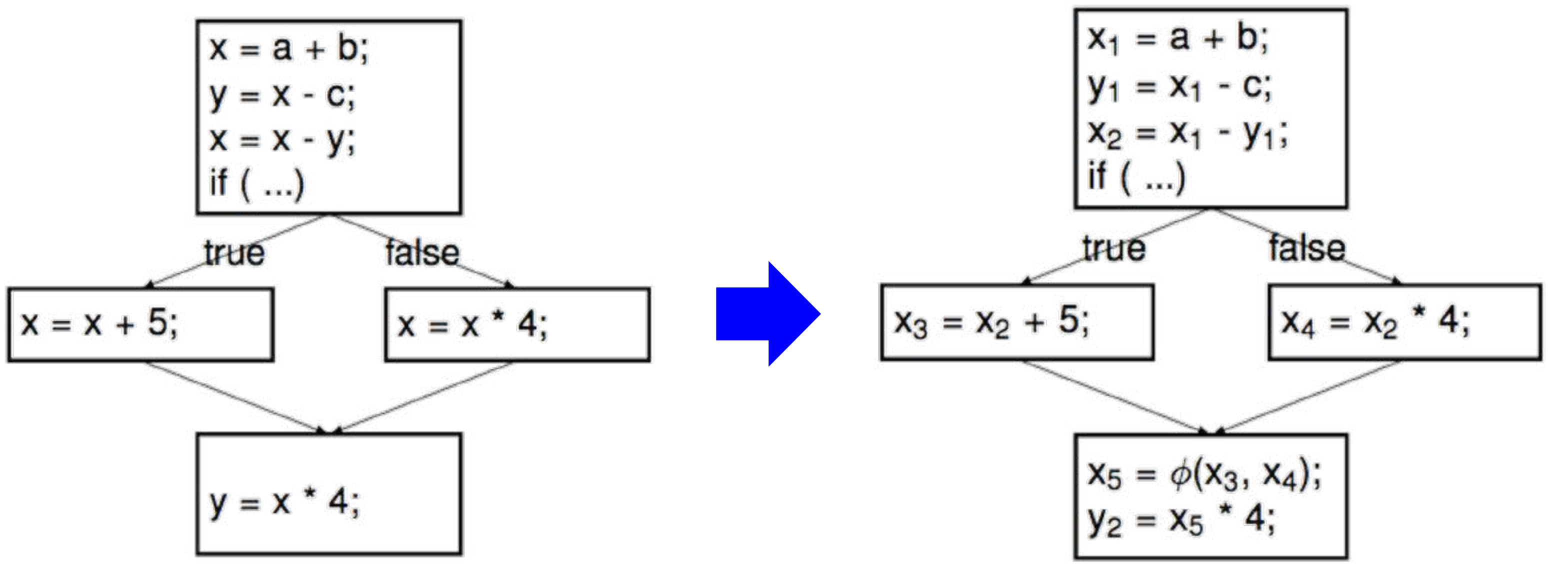
CFG for 'gcd' function

Load-and-store approach (not SSA)

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 → 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



LLVM: SSA and Phi

- All LLVM instructions are represented in the **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;
```

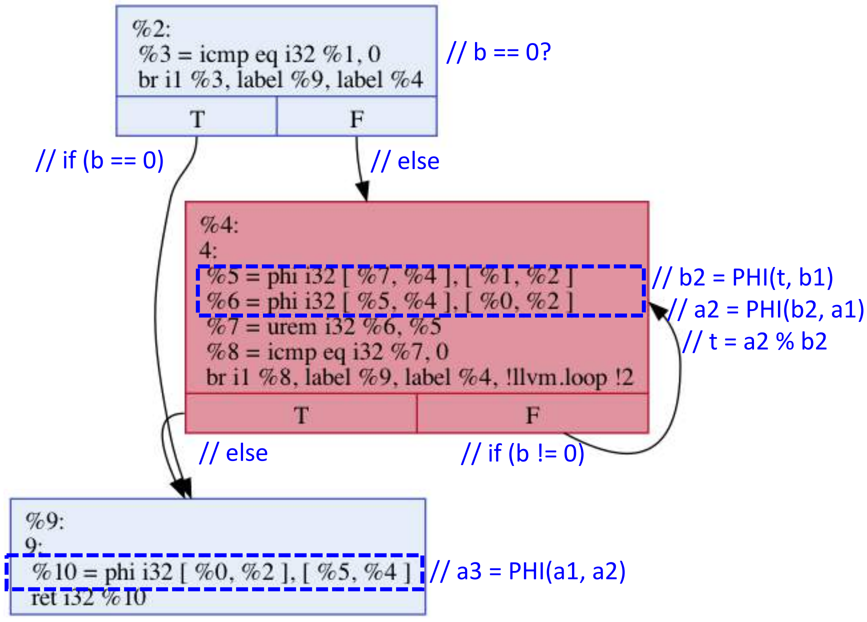
$\Rightarrow$

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

Example

- Registers
  - Unlimited #virtual registers
  - Each is written only once (SSA)
  - %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 first inst of a BB
  - The first BB cannot begin with phi
    - Has no previously executed block

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



```
%2:
 %3 = icmp eq i32 %1, 0
 br i1 %3, label %9, label %4
```

```
%4:
4:
 %5 = phi i32 [ %7, %4 ], [ %1, %2 ]
 %6 = phi i32 [ %5, %4 ], [ %0, %2 ]
 %7 = urem i32 %6, %5
 %8 = icmp eq i32 %7, 0
 br i1 %8, label %9, label %4, !llvm.loop !2
```

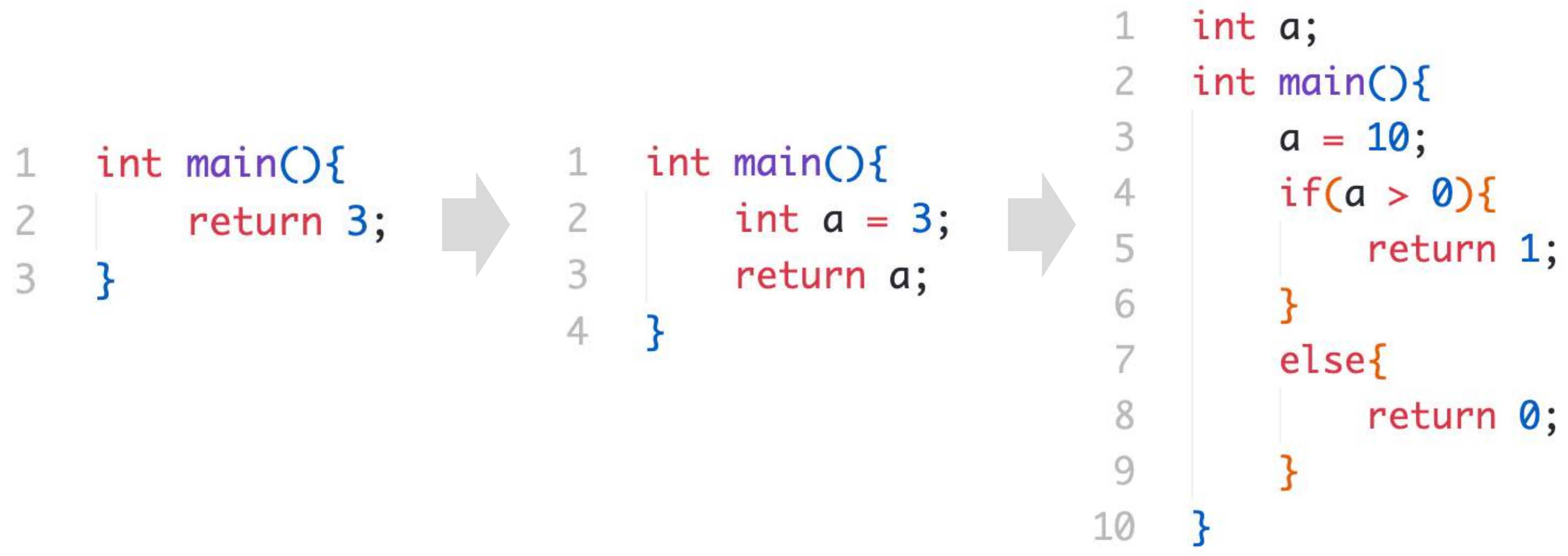
```
%9:
9:
 %10 = phi i32 [ %0, %2 ], [ %5, %4 ]
 ret i32 %10
```

CFG for 'gcd' function

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

Example: 027_if2.sysu.c

- The program
  - Global variable (int a;)
  - Variable assignment (a = 10;)
  - Binary operation (a > 0)
  - Branch (if-else)



```c
int main(){
    return 3;
}
```

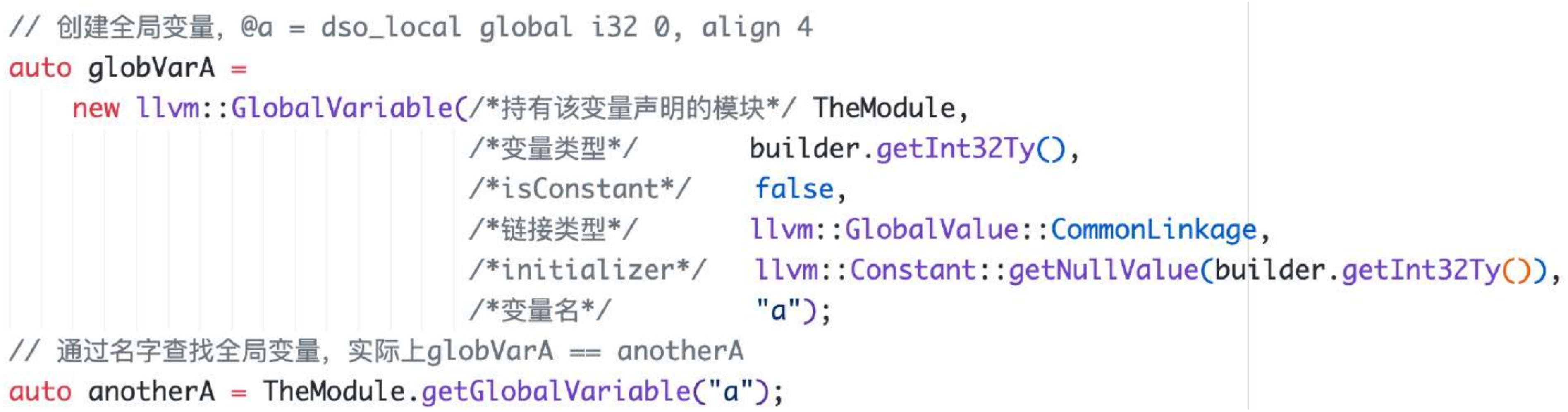
```c
int main(){
    int a = 3;
    return a;
}
```

```c
int a;
int main(){
    a = 10;
    if(a > 0){
        return 1;
    }
    else{
        return 0;
    }
}
```

Example: Global Variable

Global variable
Variable assignment
Binary operation
Branch

- Create global variable
  - Just “new” it!
  - The returned pointer is the <u>in-memory</u> representation of the global variable itself
  - If named, could be looked up in module



```
// 创建全局变量, @a = dso_local global i32 0, align 4
auto globVarA =
    new llvm::GlobalVariable(/*持有该变量声明的模块*/ TheModule,
                            /*变量类型*/         builder.getInt32Ty(),
                            /*isConstant*/      false,
                            /*链接类型*/         llvm::GlobalValue::CommonLinkage,
                            /*initializer*/     llvm::Constant::getNullValue(builder.getInt32Ty()),
                            /*变量名*/           "a");
// 通过名字查找全局变量, 实际上globVarA == anotherA
auto anotherA = TheModule.getGlobalVariable("a");
```

https://arcsysu.github.io/SYsU-lang2/#/task3_doc/ir?id=%e5%88%9b%e5%bb%ba%e5%85%a8%e5%b1%80%e5%8f%98%e9%87%8f

https://releases.llvm.org/11.0.1/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html
https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

Example: Variable Assignment & Expr

Global variable
Variable assignment
Binary operation
Branch

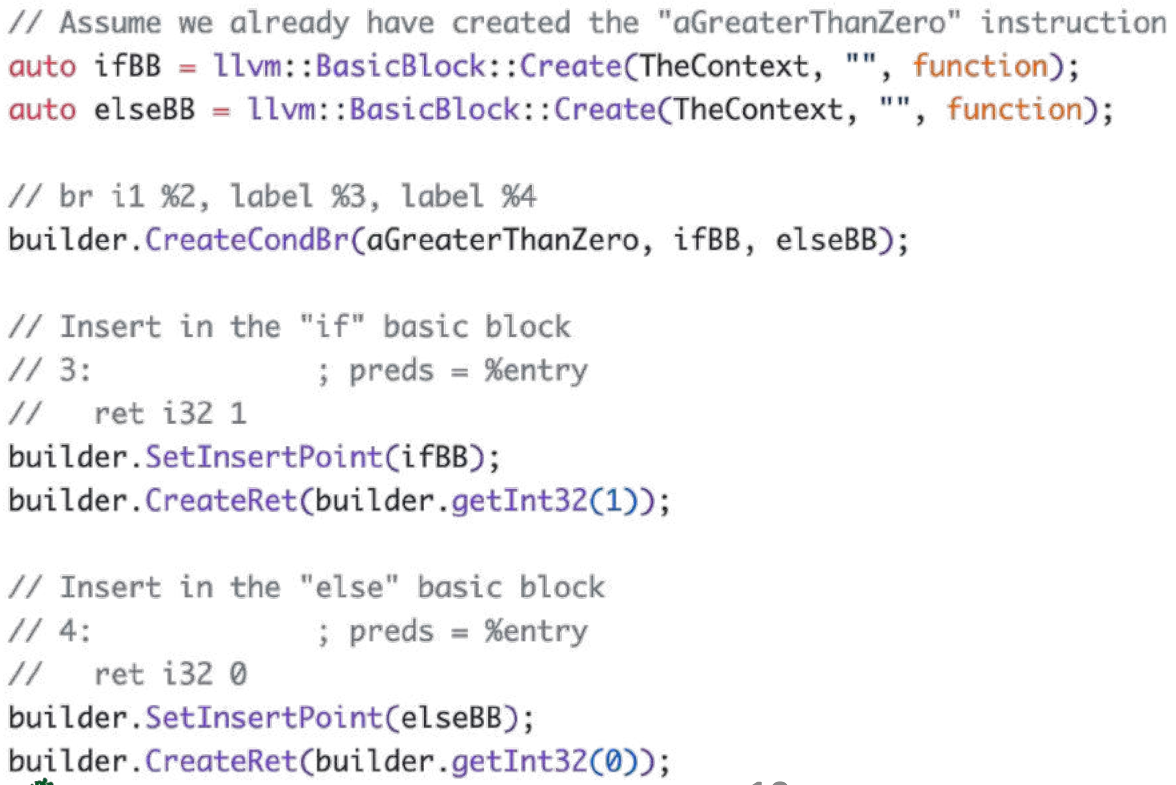
- Create assignment and operation expression
  - The "instruction" is also the "virtual register"
  - LLVM IR is strongly typed, identified by llvm::Type
  - Constant values are represented by the llvm::Constant class
    - ✗builder.CreateLoad(10, globVarA);
    - ✓builder.CreateLoad(builder.getInt32(10), globVarA);

```
// 通过名字查找全局变量
auto globVarA = TheModule.getGlobalVariable("a");
// store i32 10, ptr @a, align 4
builder.CreateStore(builder.getInt32(10), globVarA);
// %1 = load i32, ptr @a, align 4
auto localA = builder.CreateLoad(globVarA->getValueType(), globVarA);
// %2 = icmp sgt i32 %1, 0
auto aGreaterThanZero = builder.CreateICmpSGT(localA, builder.getInt32(0));
```

Example: Branching

Global variable
Variable assignment
Binary operation
Branch

- Create branch
  - Create new basic block
  - Create conditional branch
  - Change IRBuilder's insert point



```
// Assume we already have created the "aGreaterThanZero" instruction
auto ifBB = llvm::BasicBlock::Create(TheContext, "", function);
auto elseBB = llvm::BasicBlock::Create(TheContext, "", function);

// br i1 %2, label %3, label %4
builder.CreateCondBr(aGreaterThanZero, ifBB, elseBB);

// Insert in the "if" basic block
// 3:                ; preds = %entry
//   ret i32 1
builder.SetInsertPoint(ifBB);
builder.CreateRet(builder.getInt32(1));

// Insert in the "else" basic block
// 4:                ; preds = %entry
//   ret i32 0
builder.SetInsertPoint(elseBB);
builder.CreateRet(builder.getInt32(0));
```

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

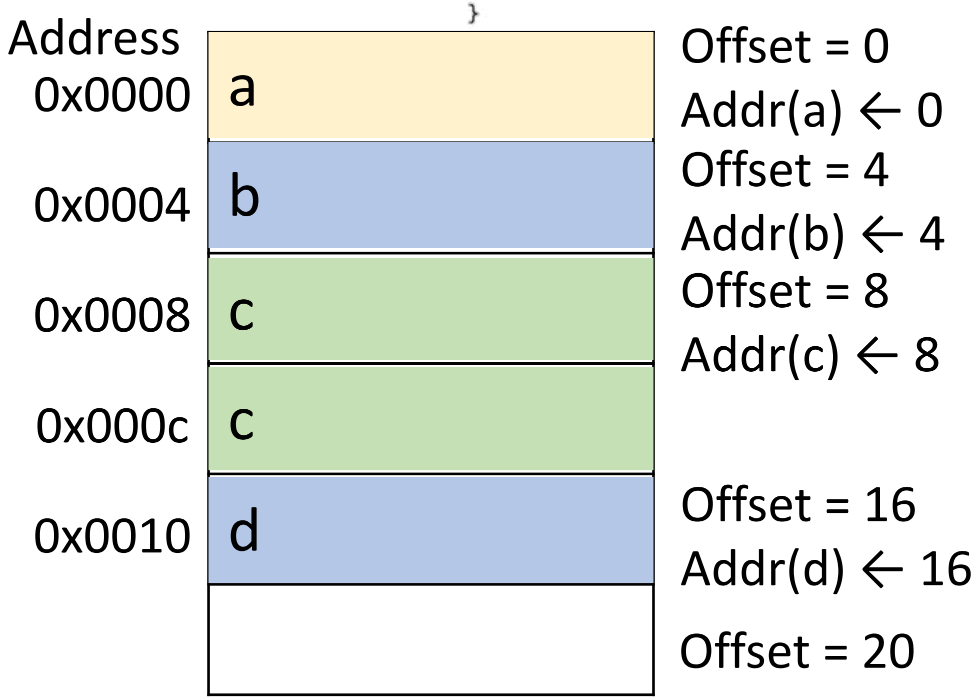
- Naive method: reserve a big memory section for all data
  - Size data section to be large enough to contain all variables
  - Location = var offset + base of data section
- Naive method wastes a lot of memory
  - Vars with limited scope need to live only briefly in memory
    - E.g. function variables 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 <u>sequentially</u>
  - Current <u>offset</u> 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 | float
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* -> float { *B.type = float; 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:
  - *float 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: float 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 -> float { $B.type = float; B.width = 8;$ }

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: float x; int i;

$\uparrow$

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;* }

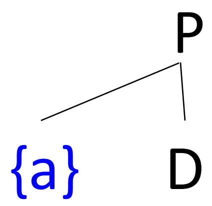
⑦ B -> float { *B.type = float; B.width = 8;* }

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: float 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;* }

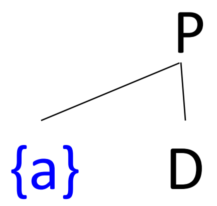
⑦ B -> float { *B.type = float; B.width = 8;* }

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: float 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 -> float { *B.type = float; B.width = 8;* }

⑧ C -> ε { *C.type = t; C.width = w;* }

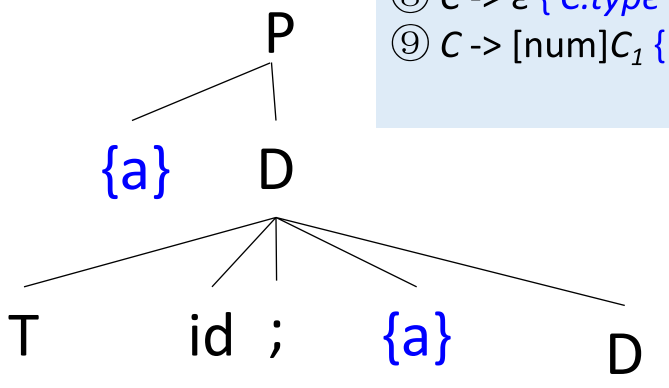
⑨ C -> [num]C_1 { *C.type = array(num.val,* C_1*.type);*
*C.width = num.val ** C_1*.width;* }

offset = 0

Example

- Input: float 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* -> float { *B.type = float; 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;* }

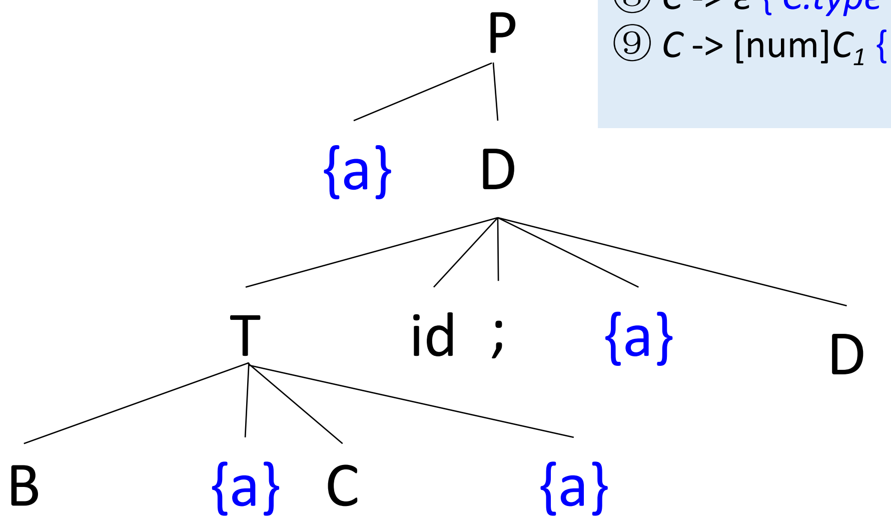


offset = 0

Example

- Input: float 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 -> float { *B.type = float; 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: float 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 { $T.type = C.type; T.width = C.width;$ }

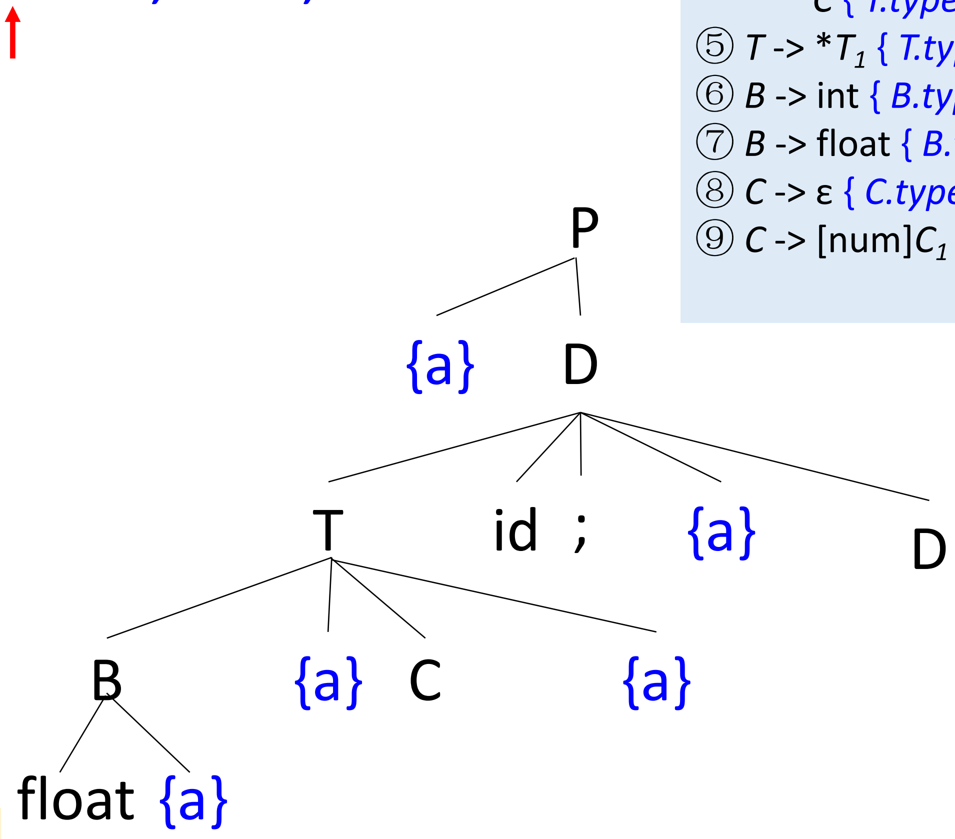
⑤ T -> $*T_1$ { $T.type = pointer(T_1.type); T.width = 4;$ }

⑥ B -> int { $B.type = int; B.width = 4;$ }

⑦ B -> float { $B.type = float; B.width = 8;$ }

⑧ C -> ε { $C.type = t; C.width = w;$ }

⑨ C -> [num]C_1 { $C.type = array(num.val, C_1.type);$
$C.width = num.val * C_1.width;$ }

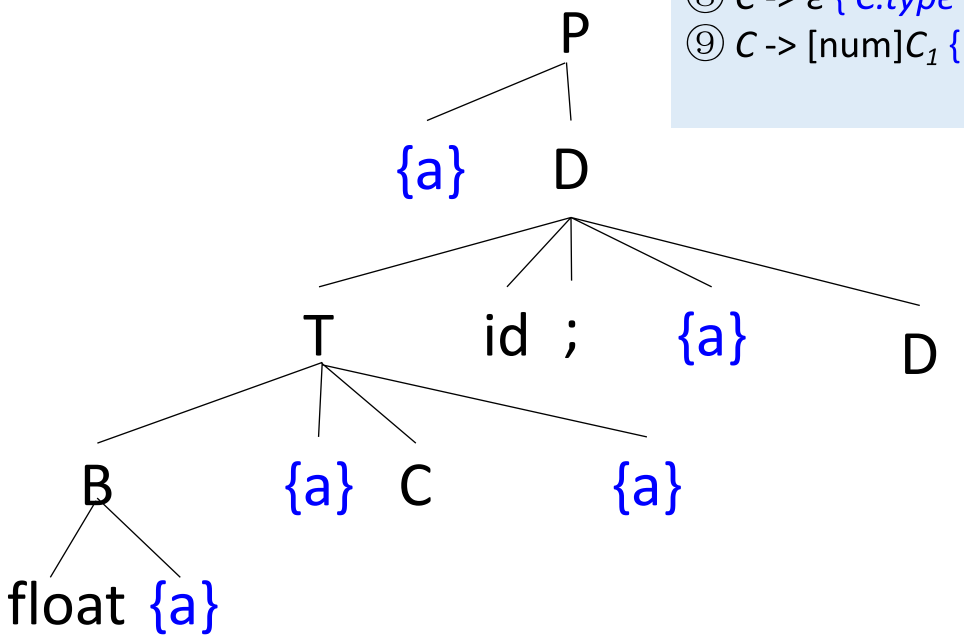


offset = 0

Example

- Input: float 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* -> float { *B.type = float; 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;* }

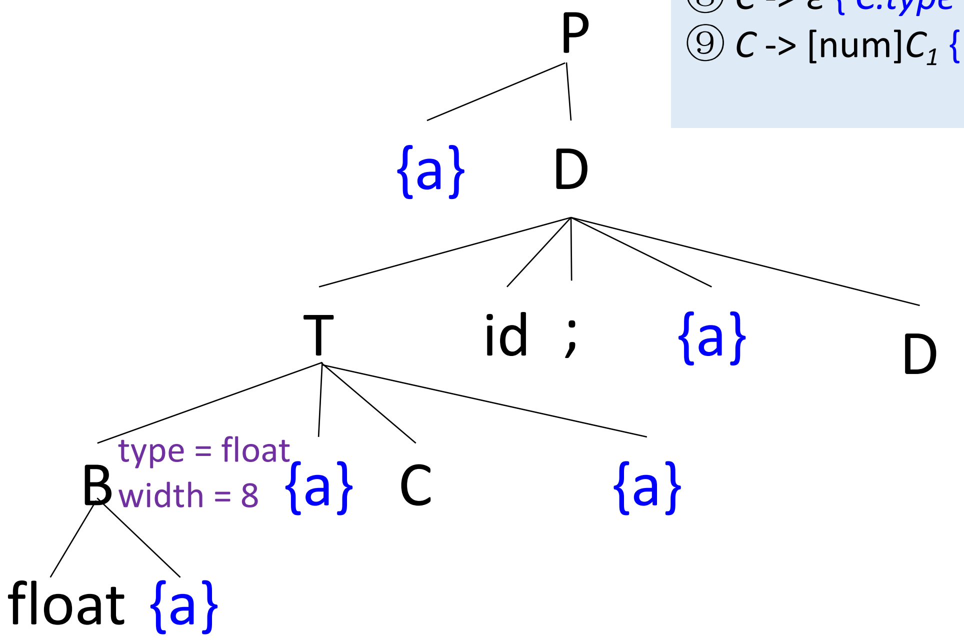


offset = 0

Example

- Input: float 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 -> float { *B.type = float; 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;* }

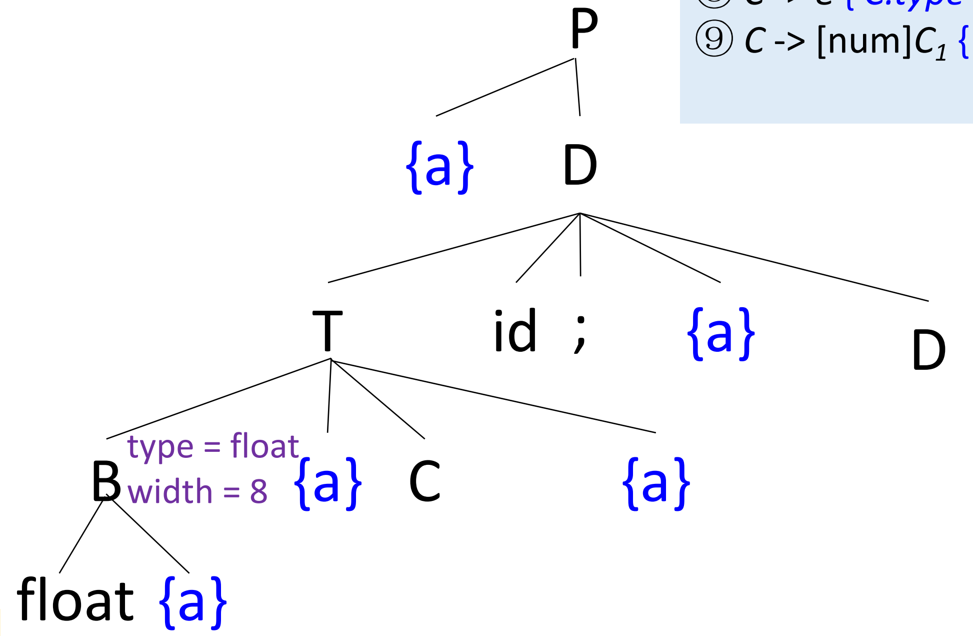


offset = 0

Example

- Input: float 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 -> float { $B.type = float; B.width = 8;$ }
⑧ C -> ε { $C.type = t; C.width = w;$ }
⑨ C -> [num]C_1 { $C.type = array(num.val, C_1.type);$
 $C.width = num.val * C_1.width;$ }



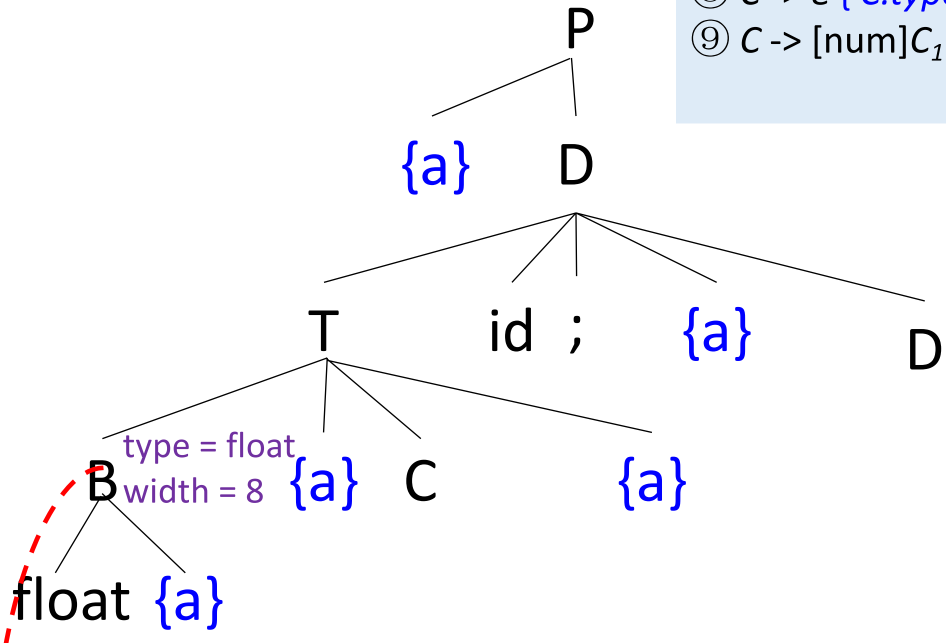
offset = 0

t = float
w = 8

Example

- Input: float 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* -> float { *B.type = float; 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;* }



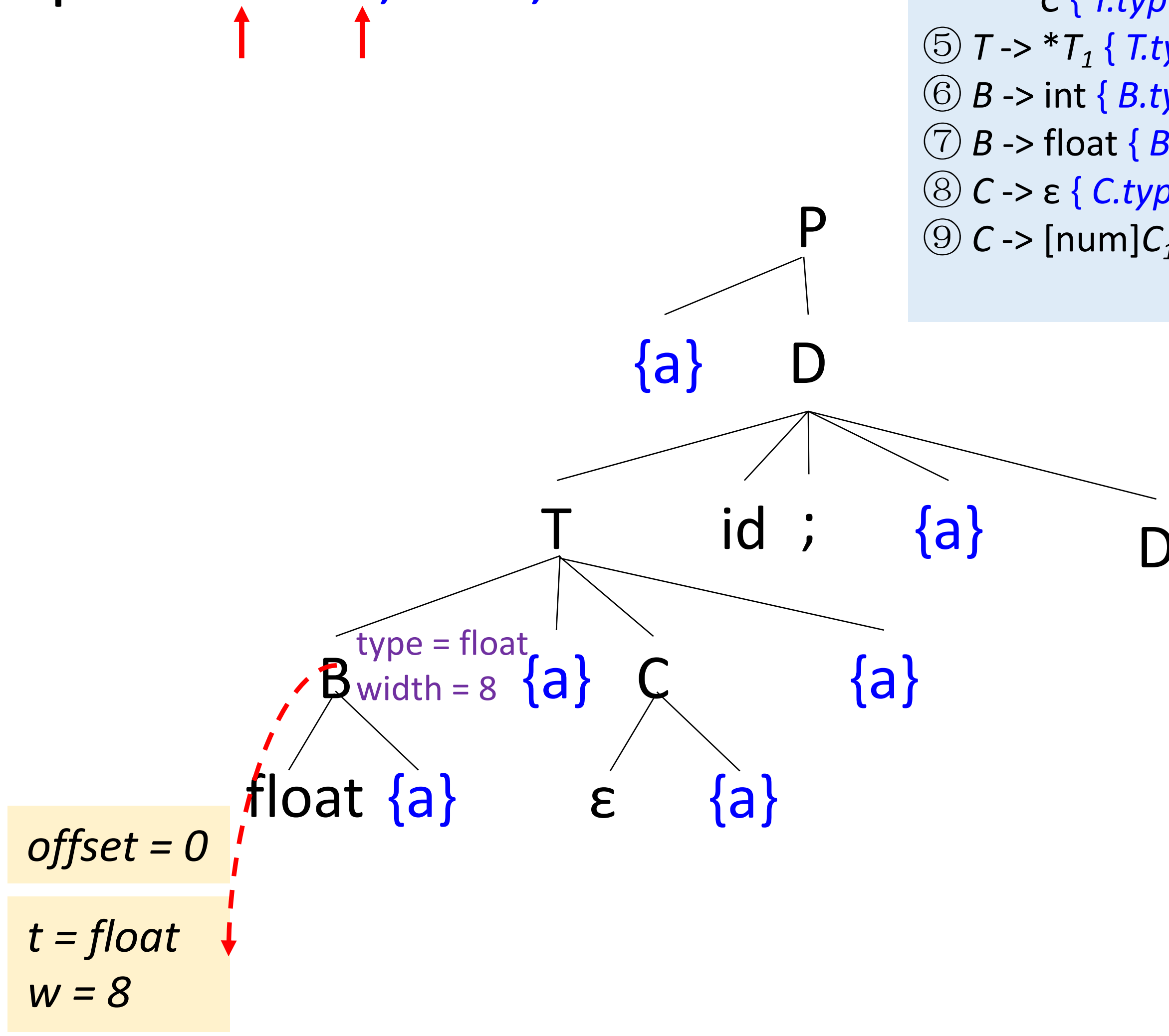
offset = 0

t = float
w = 8

Example

- Input: float x; int i;

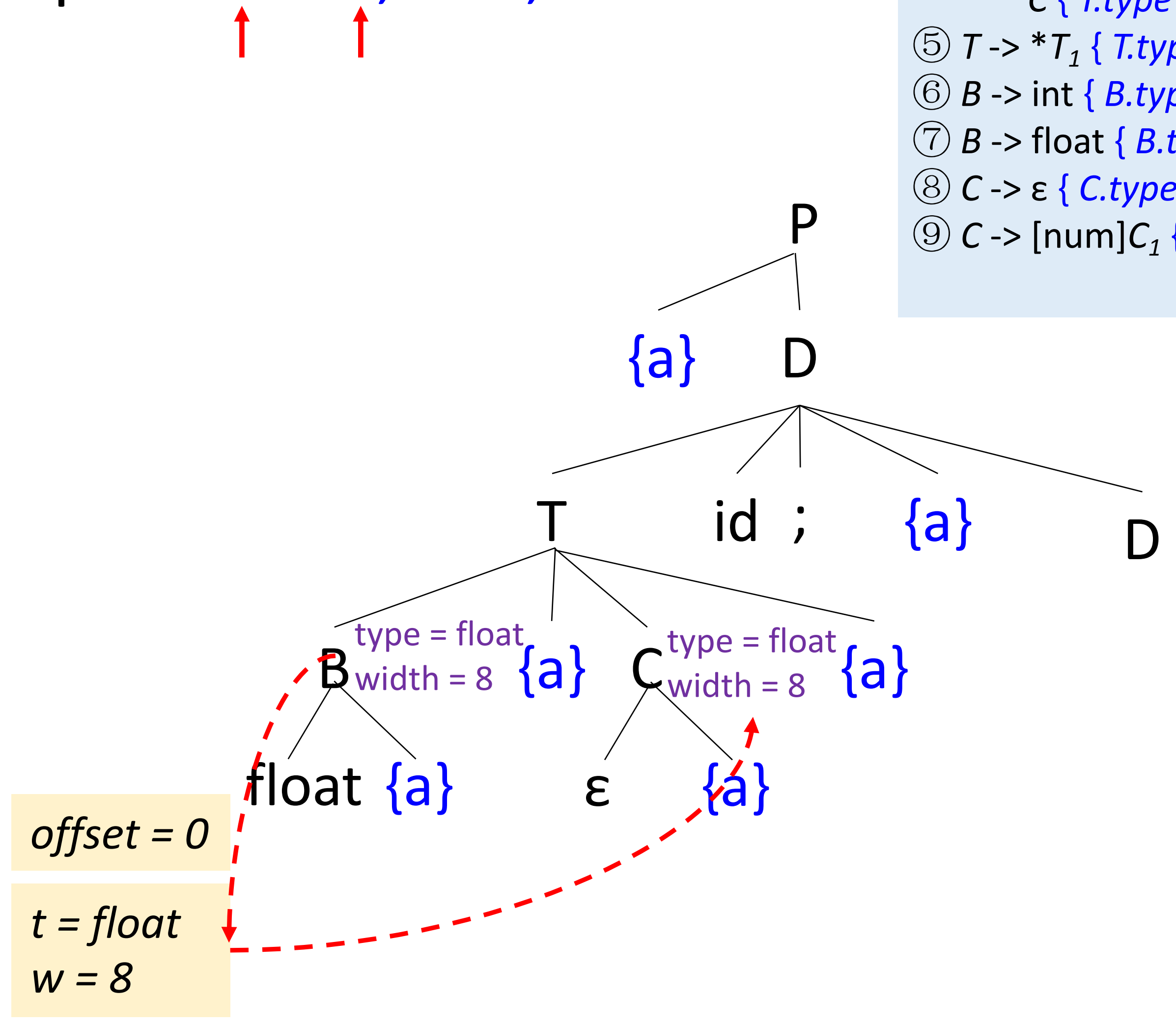
① P -> { $offset = 0$ } D
② D -> T id; { $enter(id.lexeme, T.type, offset);$ $offset = offset + T.width;$ } D_1
③ D -> ε
④ T -> B { $t = B.type; w = B.width;$ } C { $T.type = C.type; T.width = C.width;$ }
⑤ T -> *T_1 { $T.type = pointer(T_1.type); T.width = 4;$ }
⑥ B -> int { $B.type = int; B.width =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

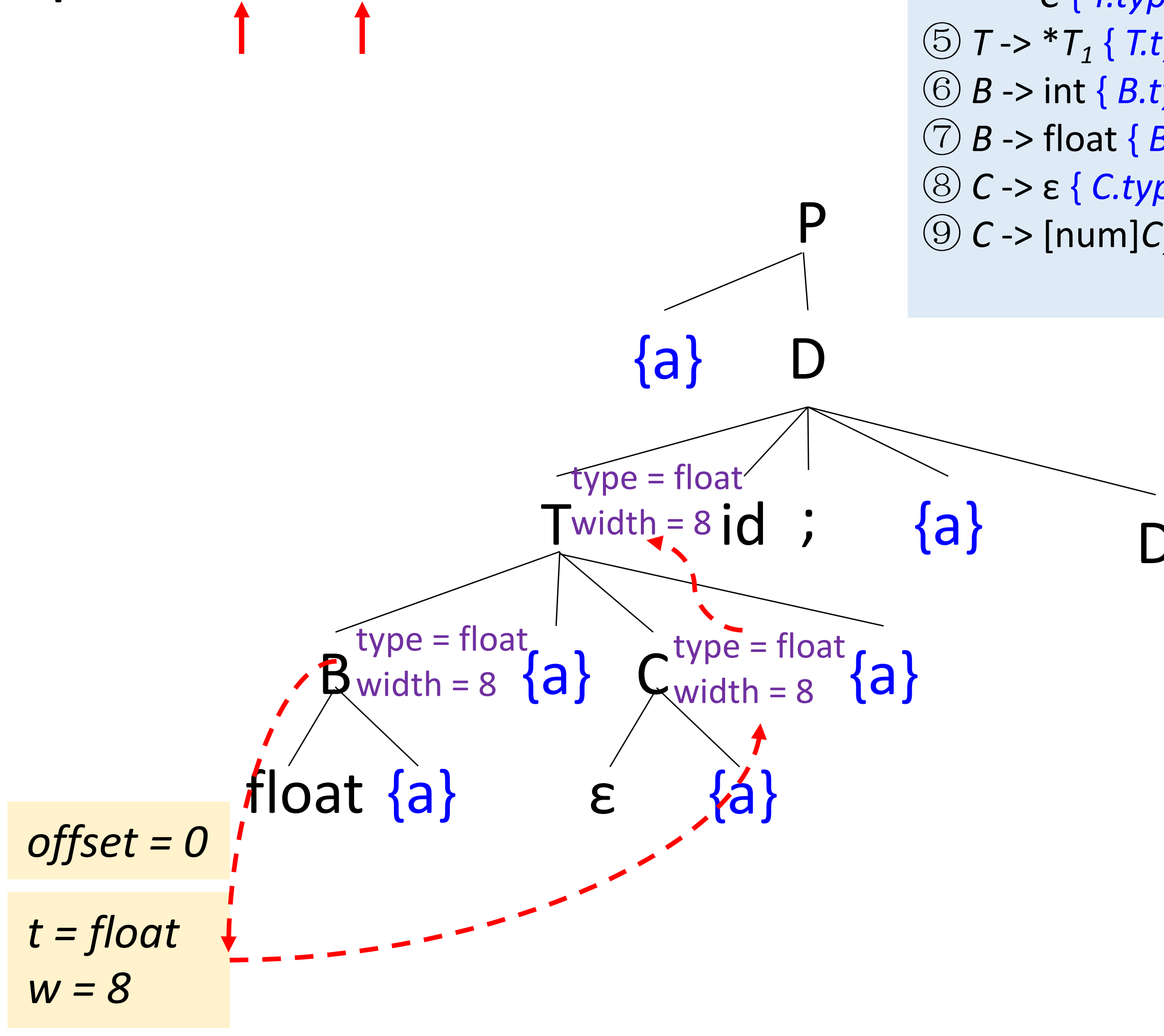
⑤ T -> $*T_1$ { *T.type = pointer(T_1.type); T.width = 4;* }

⑥ B -> int { *B.type = int; B.width = 4;* }

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

⑧ C -> ε { *C.type = t; C.width = w;* }

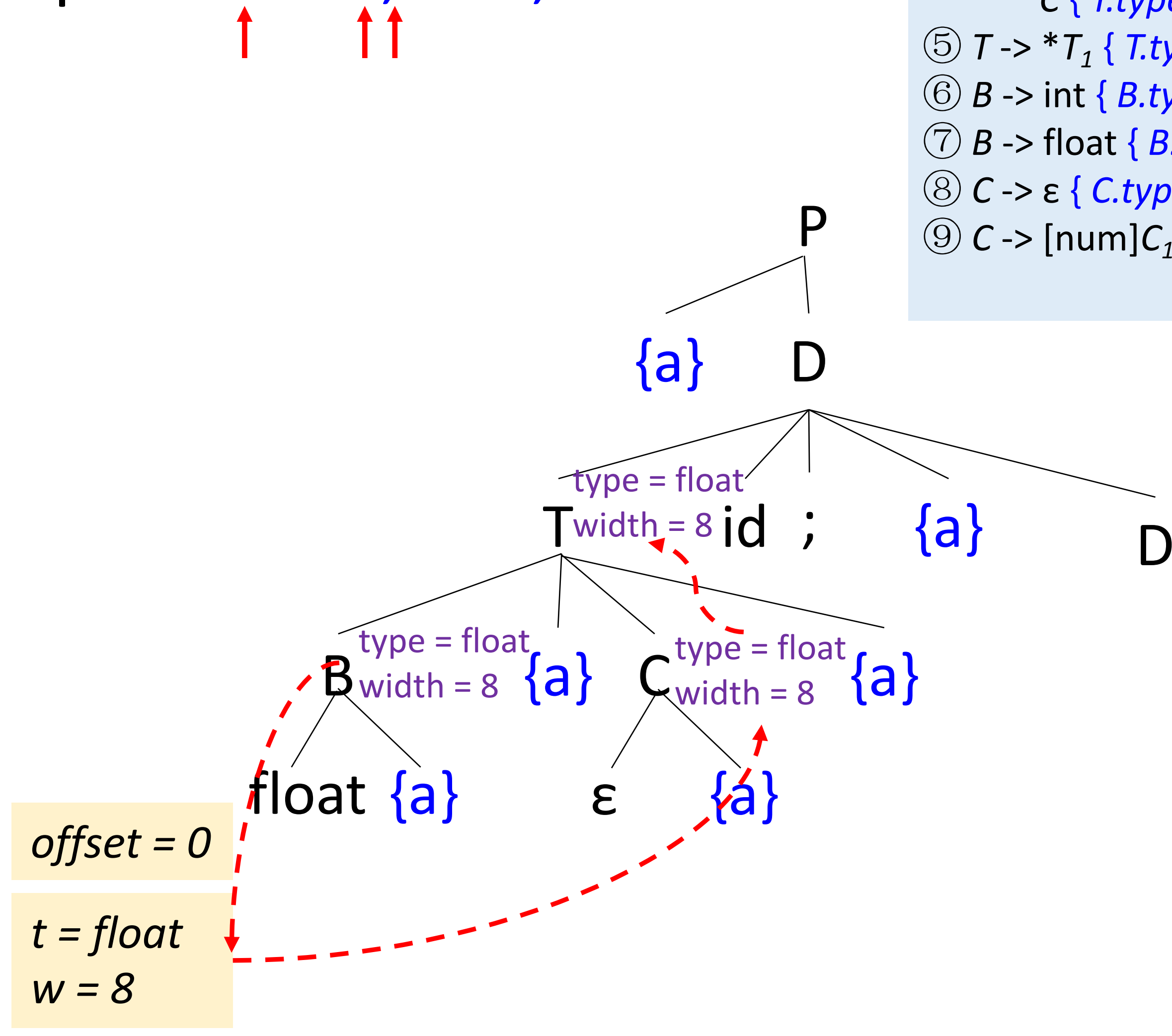
⑨ C -> [num]C_1 { *C.type = array(num.val, C_1.type);*
*C.width = num.val * C_1.width;* }



Example

- Input: float 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 -> float { *B.type = float; 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;* }

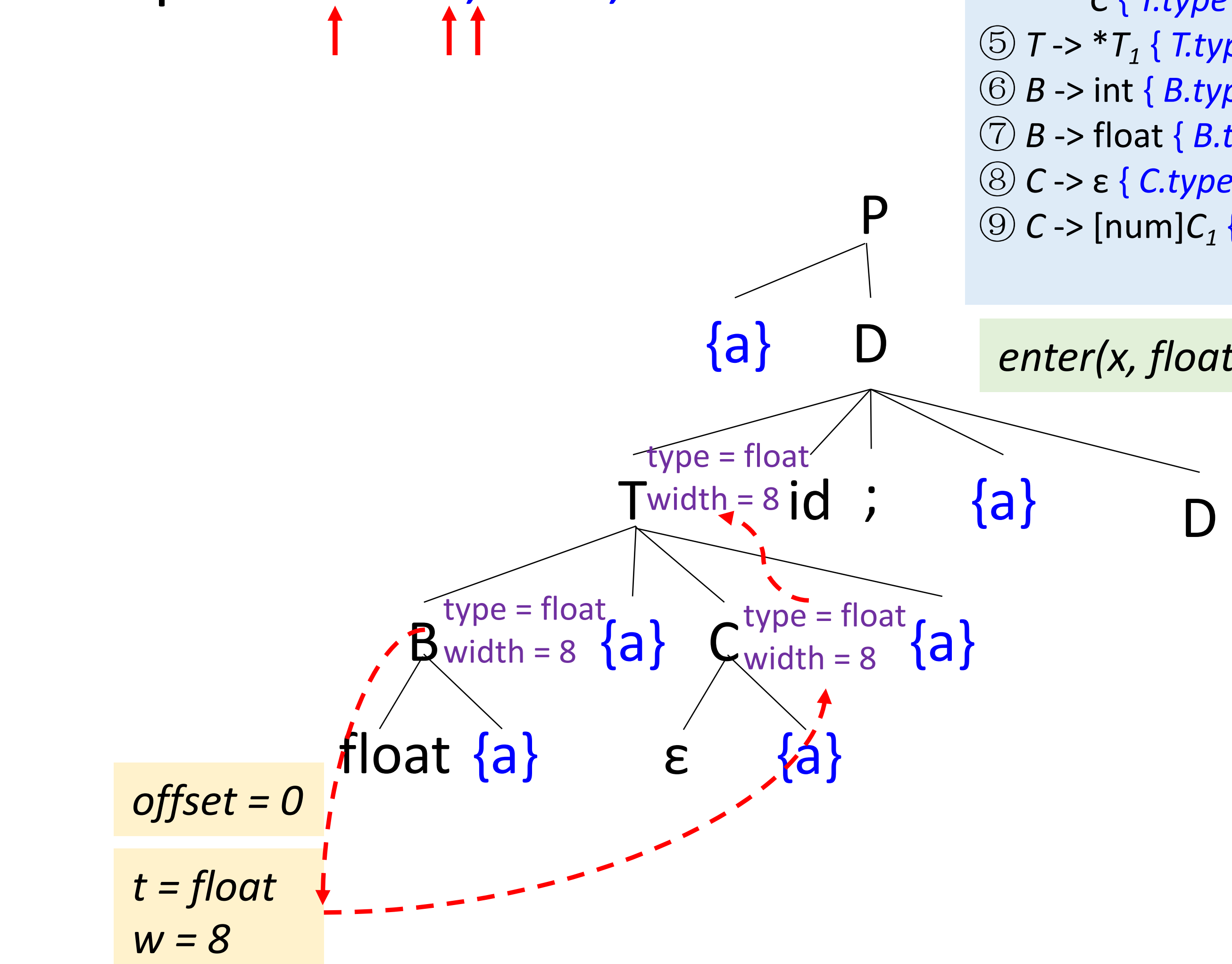


Example

- Input: float 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 -> float { $B.type = float; B.width = 8;$ }
⑧ C -> ε { $C.type = t; C.width = w;$ }
⑨ C -> [num]C_1 { $C.type = array(num.val, C_1.type);$ $C.width = num.val * C_1.width;$ }

enter(x, float, 0)



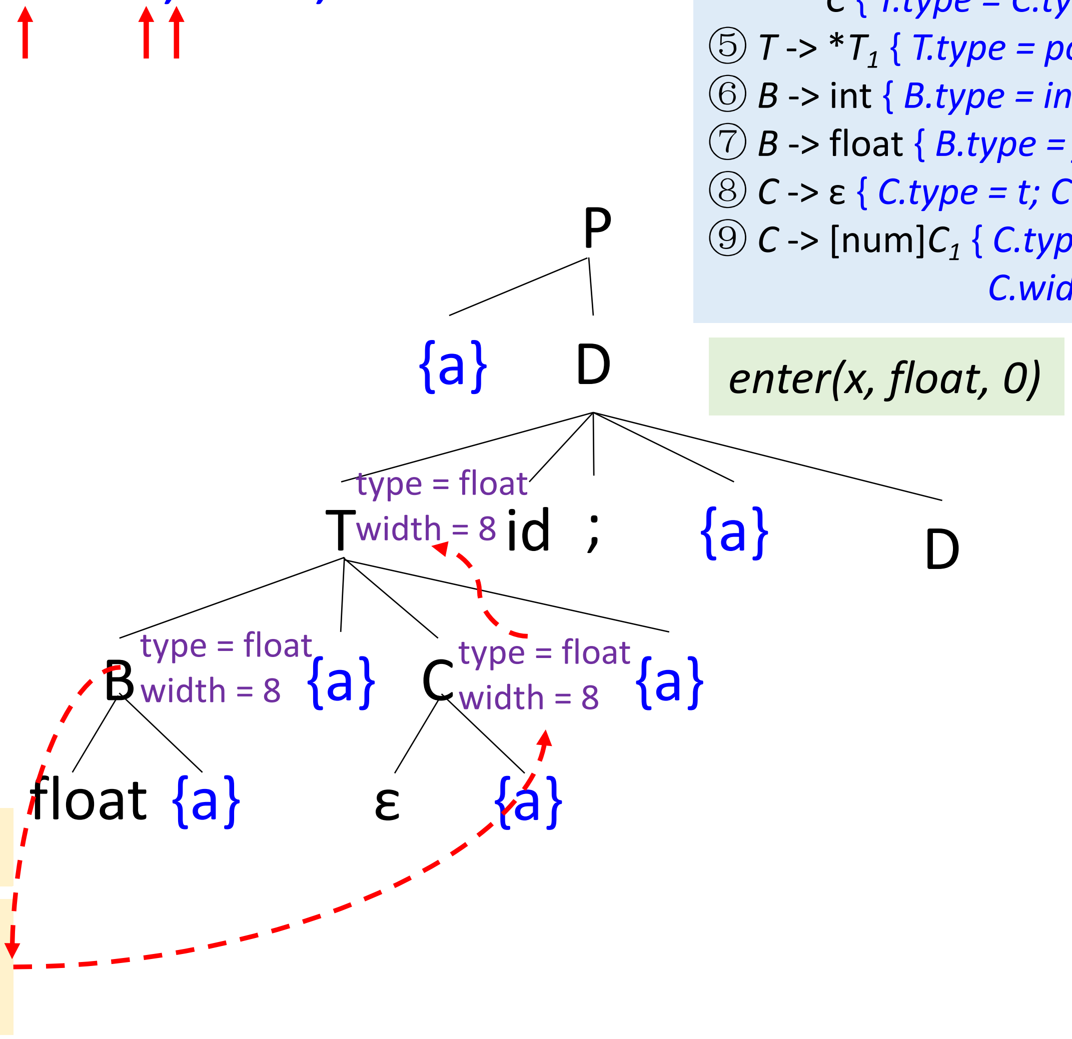
offset = 0

t = float
w = 8

Example

- Input: float x; int i;

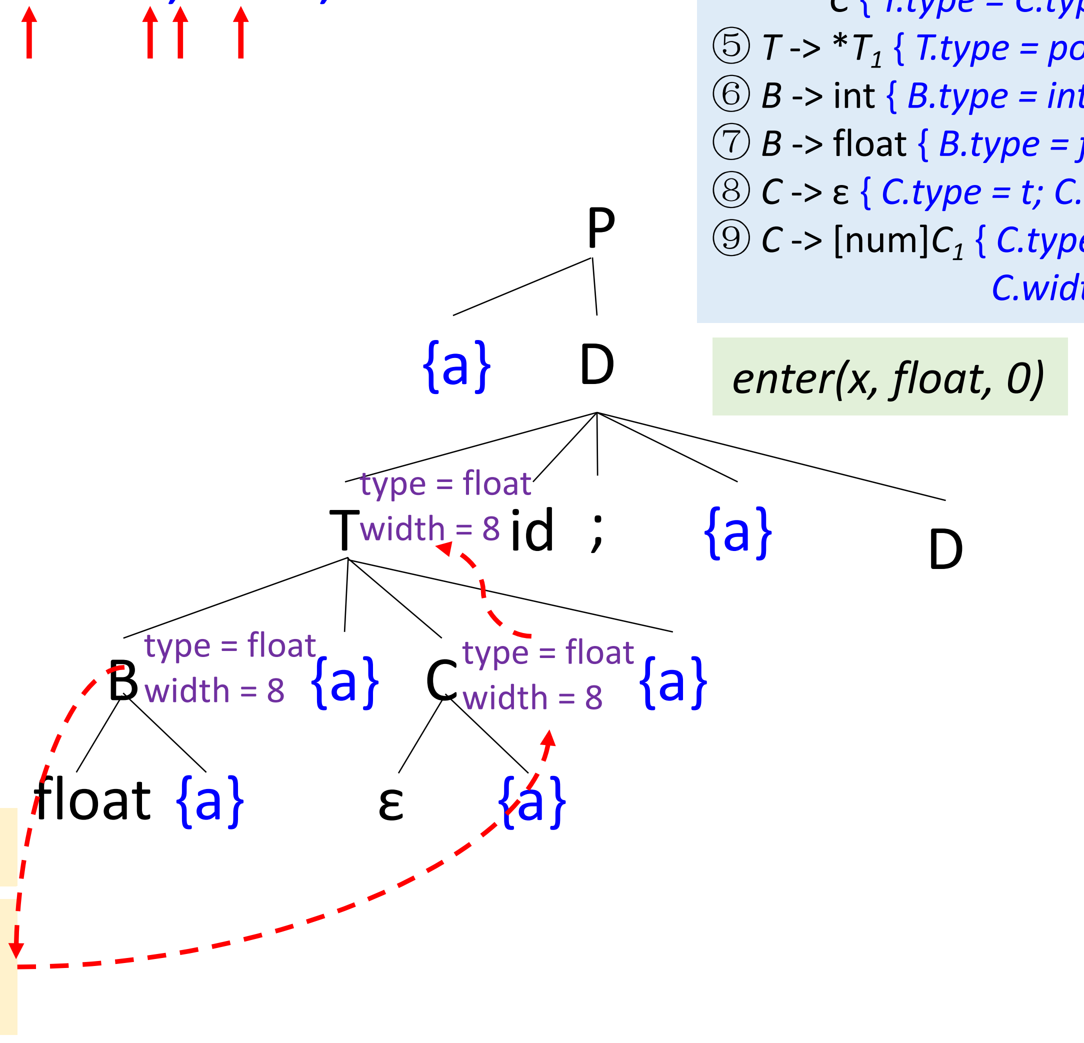
① *P* -> { *offset = 0* } *D*
② *D* -> *T* id; { *enter(id.lexeme, T.type, offset);*
offset = offset + T.width; } D_1
③ *D* -> ε
④ *T* -> *B* { *t = B.type; w = B.width;* }
C { *T.type = C.type; T.width = C.width;* }
⑤ *T* -> *T_1 { *T.type = pointer(* T_1*.type); T.width = 4;* }
⑥ *B* -> int { *B.type = int; B.width = 4;* }
⑦ *B* -> float { *B.type = float; 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: float 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 -> float { *B.type = float; 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;* }

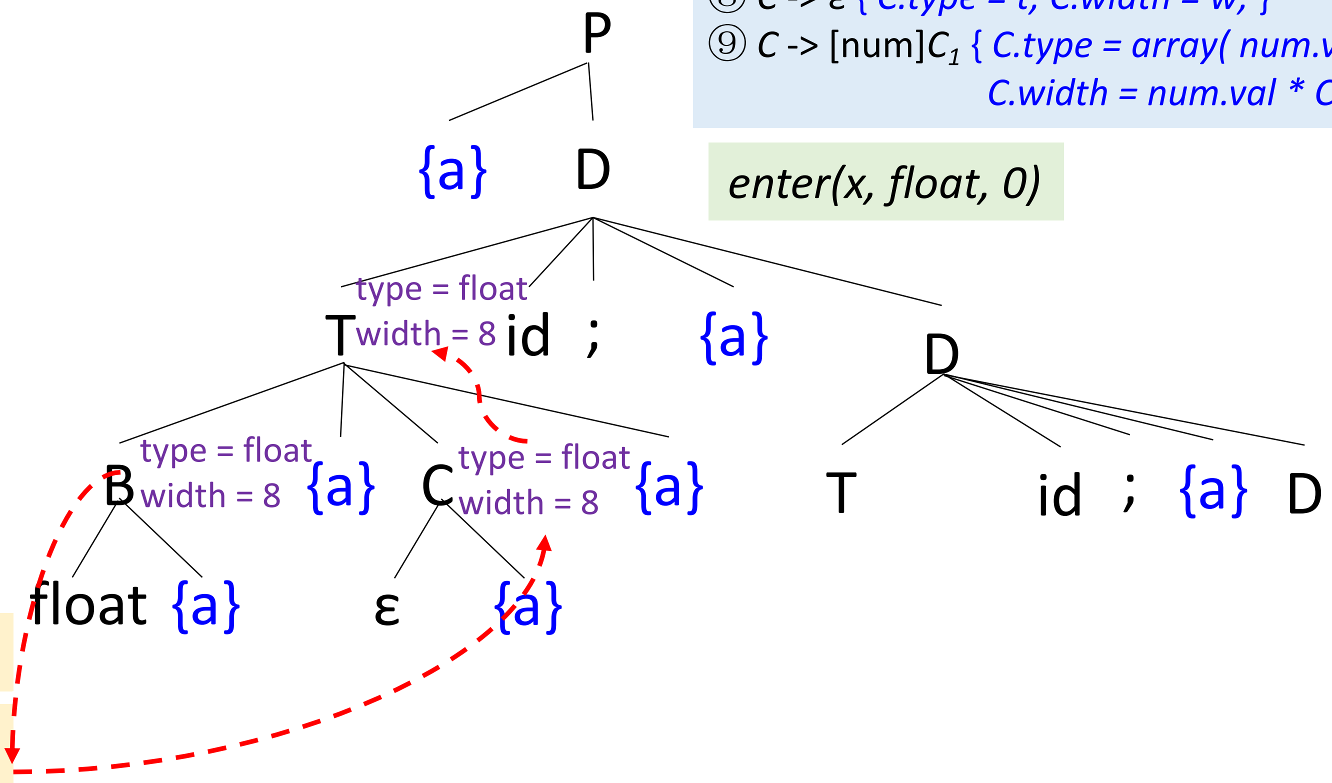


Example

- Input: float 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 -> float { B.type = float; 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; }

enter(x, float, 0)



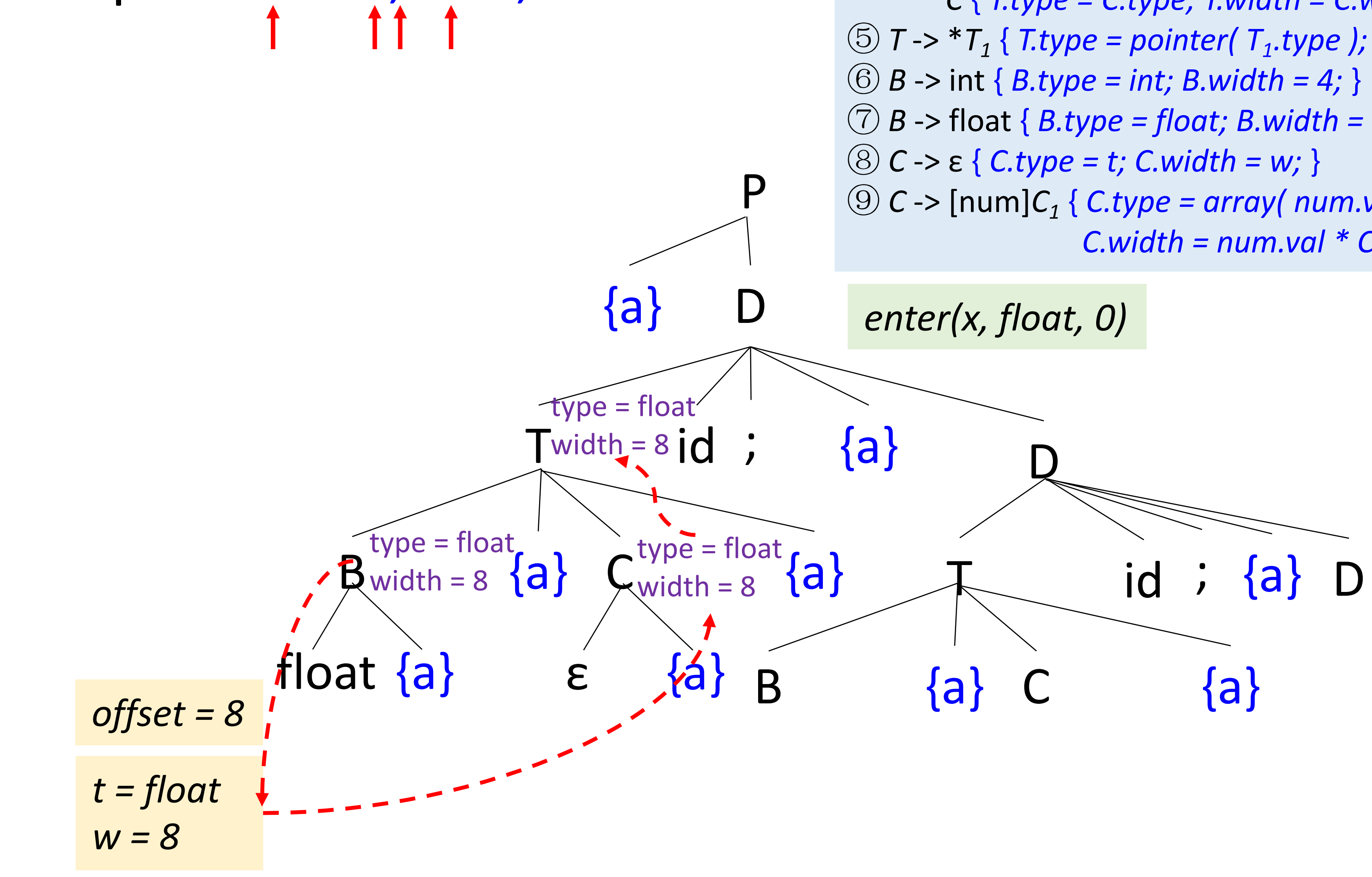
offset = 8

t = float
w = 8

Example

- Input: float x; int i;

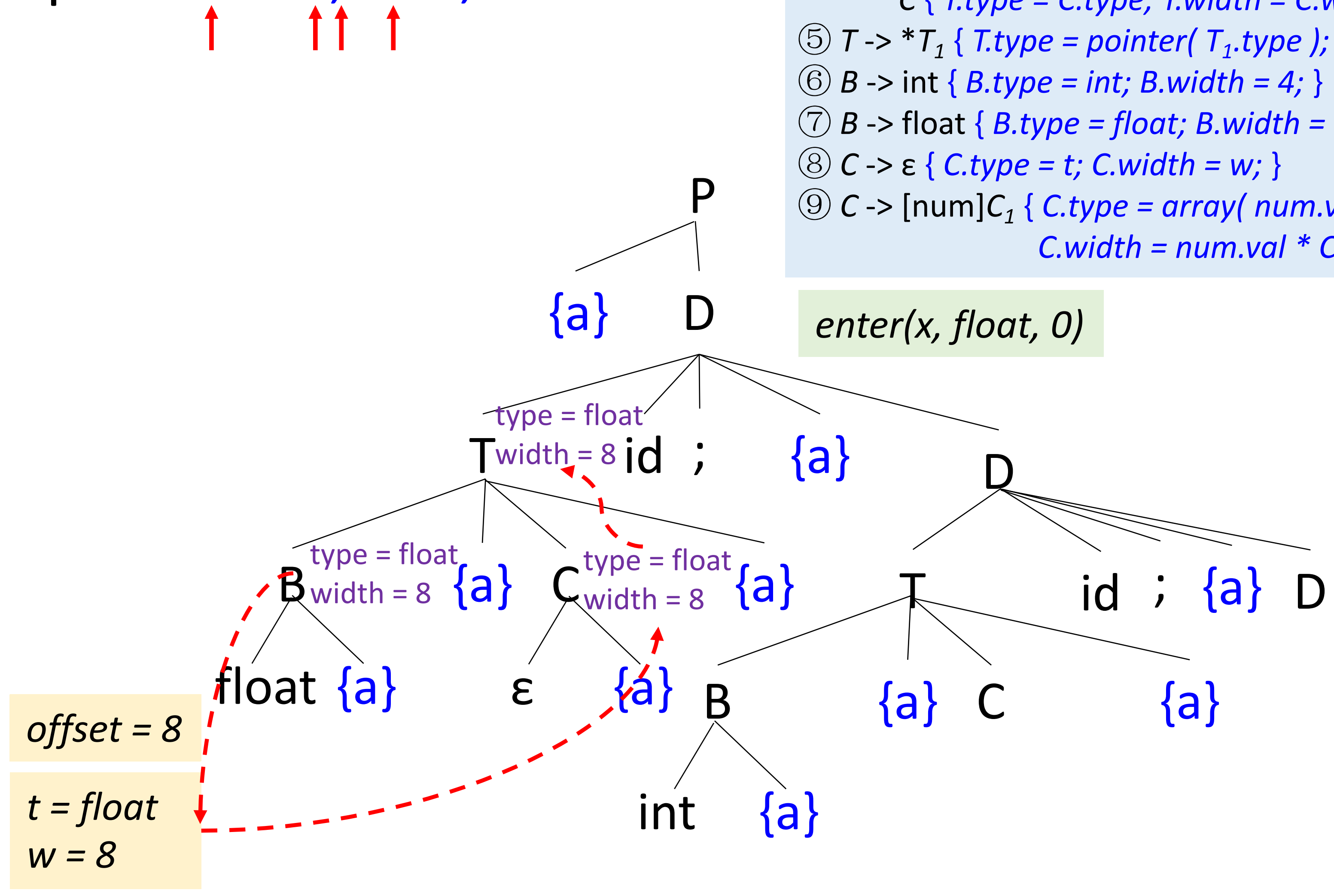
① P -> { *offset = 0* } D
② D -> T id; { *enter(id.lexeme, T.type, offset);*
offset = offset + T.width; } D_1
③ D -> ε
④ T -> B { *t = B.type; w = B.width;* }
C { *T.type = C.type; T.width = C.width;* }
⑤ T -> *T_1 { *T.type = pointer(T_1.type); T.width = 4;* }
⑥ B -> int { *B.type = int; B.width = 4;* }
⑦ B -> float { *B.type = float; 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: float x; int i;

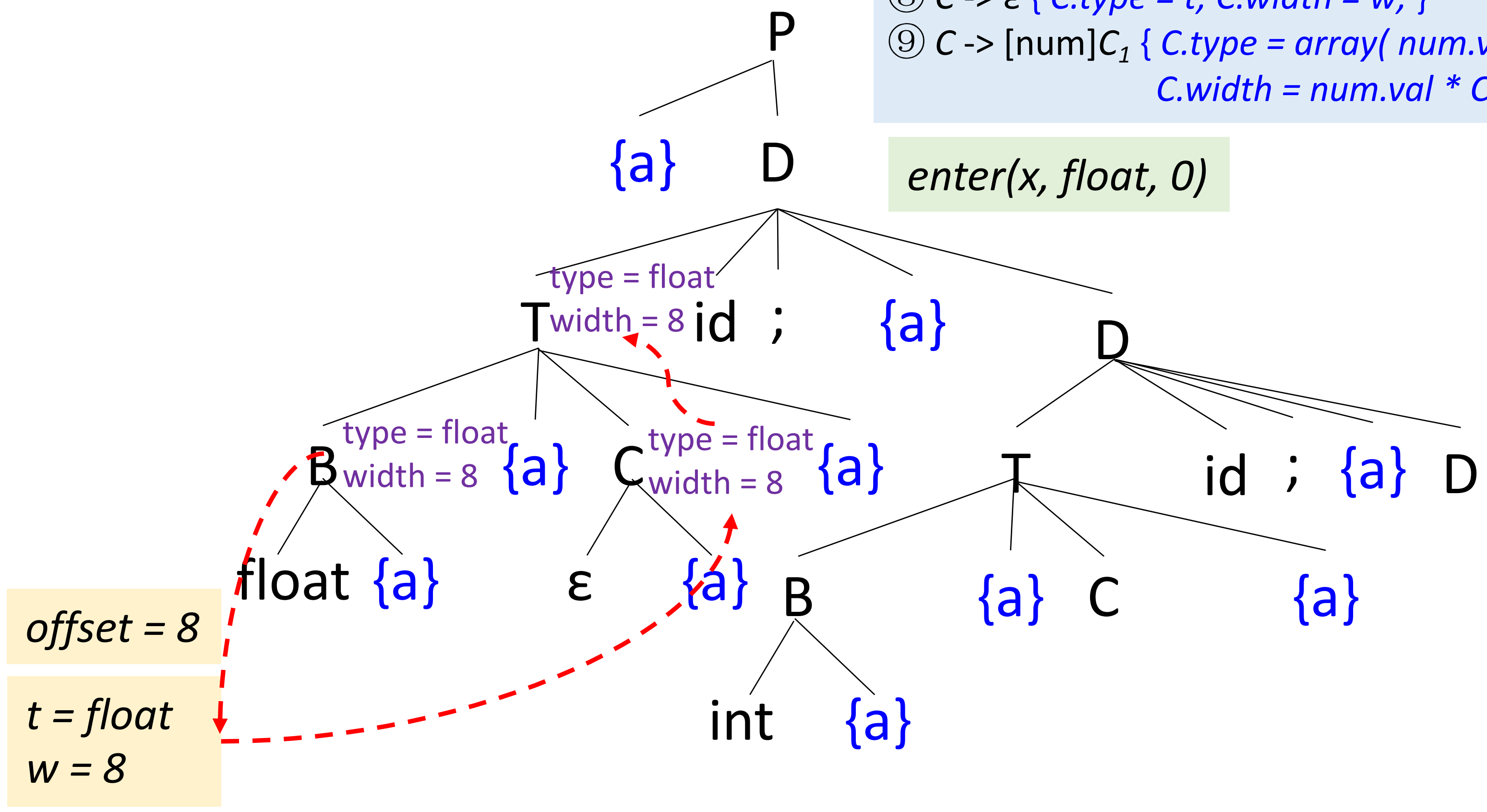
① P -> { $offset = 0$ } D
② D -> T id; { $enter(id.lexeme, T.type, offset);$
$offset = offset + T.width;$ } D_1
③ D -> ε
④ T -> B { $t = B.type; w = B.width;$ }
C { $T.type = C.type; T.width = C.width;$ }
⑤ T -> *T_1 { $T.type = pointer(T_1.type); T.width = 4;$ }
⑥ B -> int { $B.type = int; B.width =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

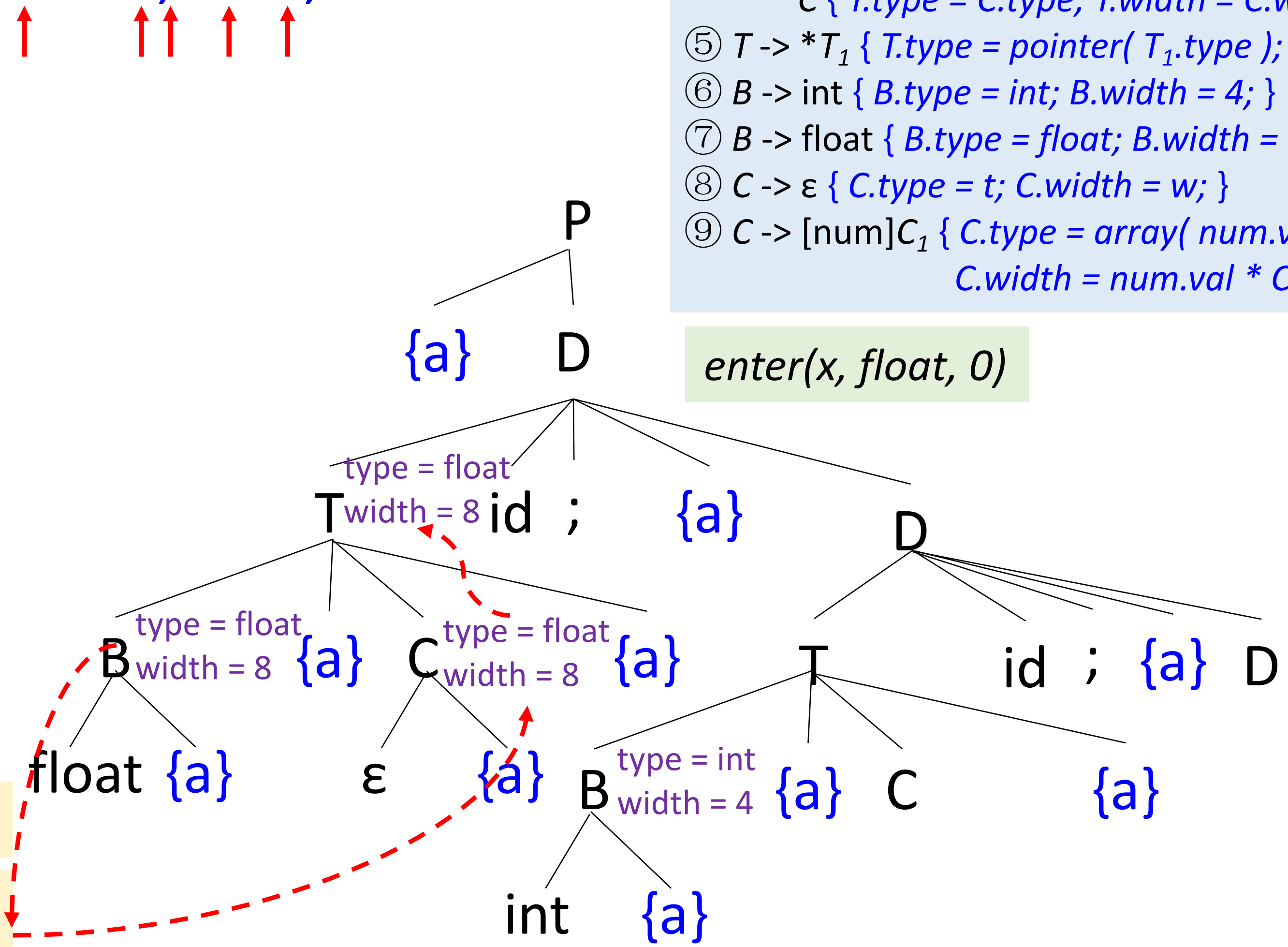
⑥ B -> int { *B.type = int; B.width = 4;* }

⑦ B -> float { *B.type = float; 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;* }

enter(x, float, 0)



offset = 8

t = float
w = 8

Example

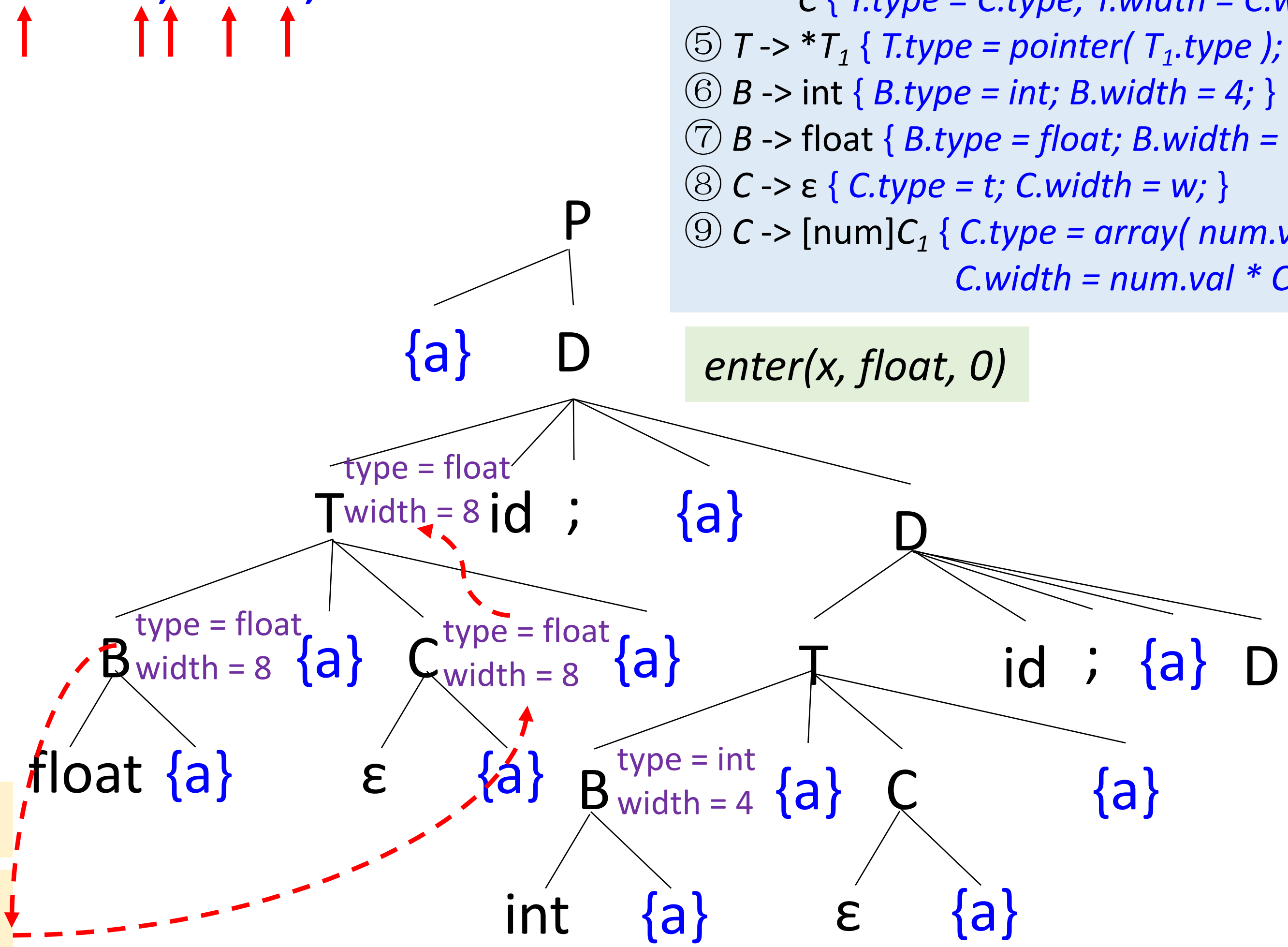
- Input: float x; int i;

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 -> float { *B.type = float; 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;* }

enter(x, float, 0)

offset = 8

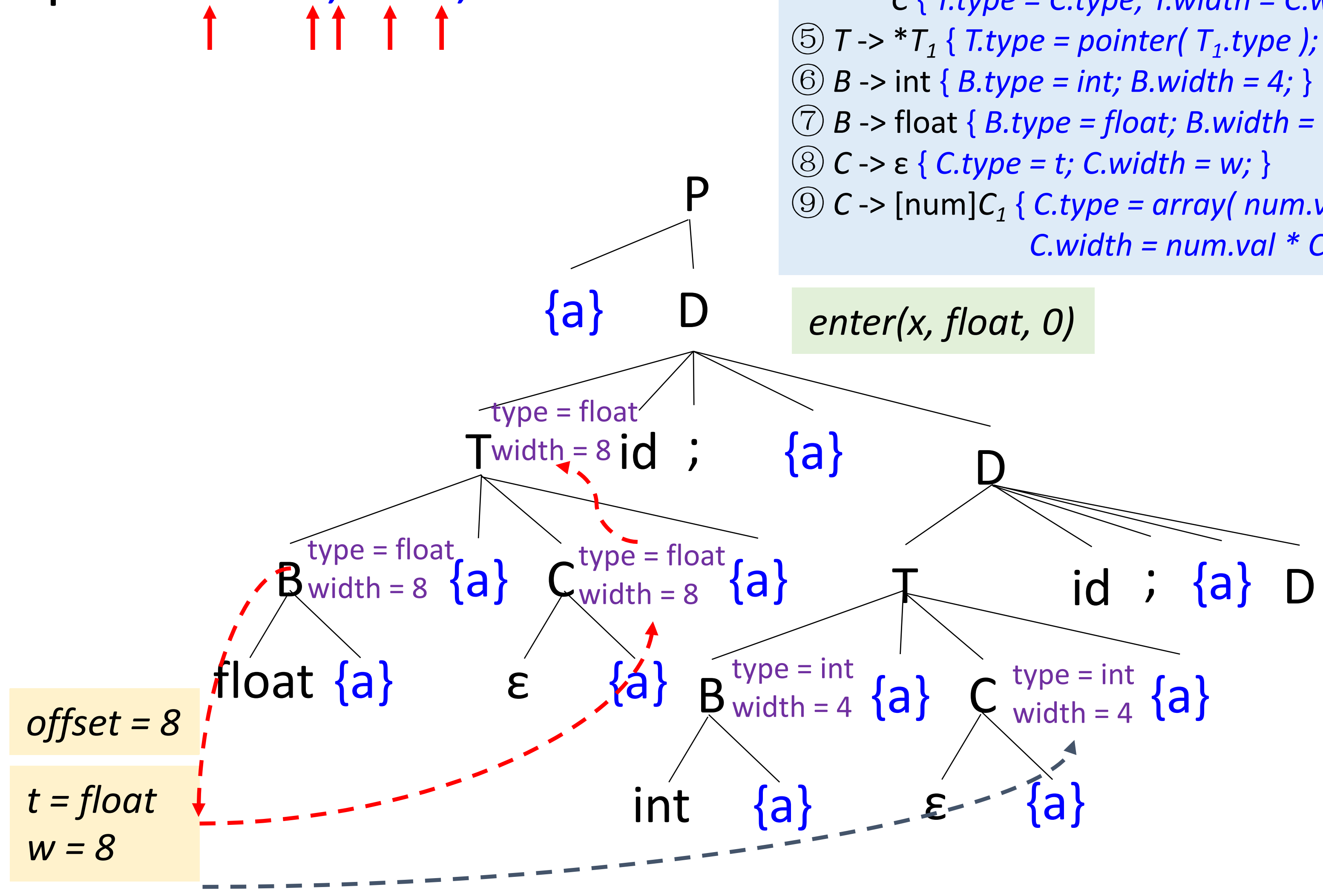
t = float
w = 8



Example

- Input: float 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 $B \to$ float { *B.type = float; 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;* }



Example

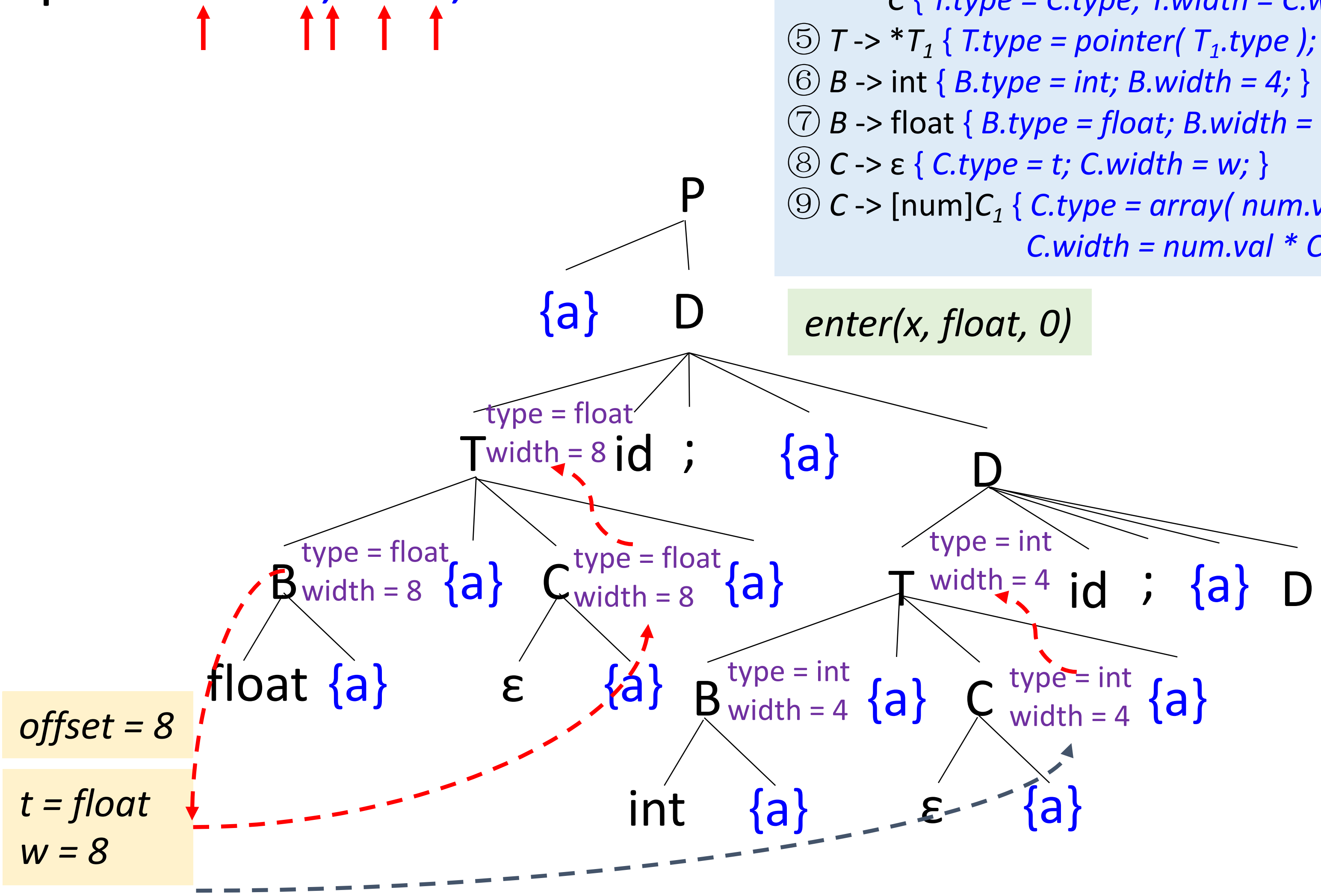
- Input: float x; int i;

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 -> float { $B.type = float; B.width = 8;$ }
⑧ C -> ε { $C.type = t; C.width = w;$ }
⑨ C -> [num]C_1 { $C.type = array(num.val, C_1.type);$
$C.width = num.val * C_1.width;$ }

enter(x, float, 0)

offset = 8

t = float
w = 8

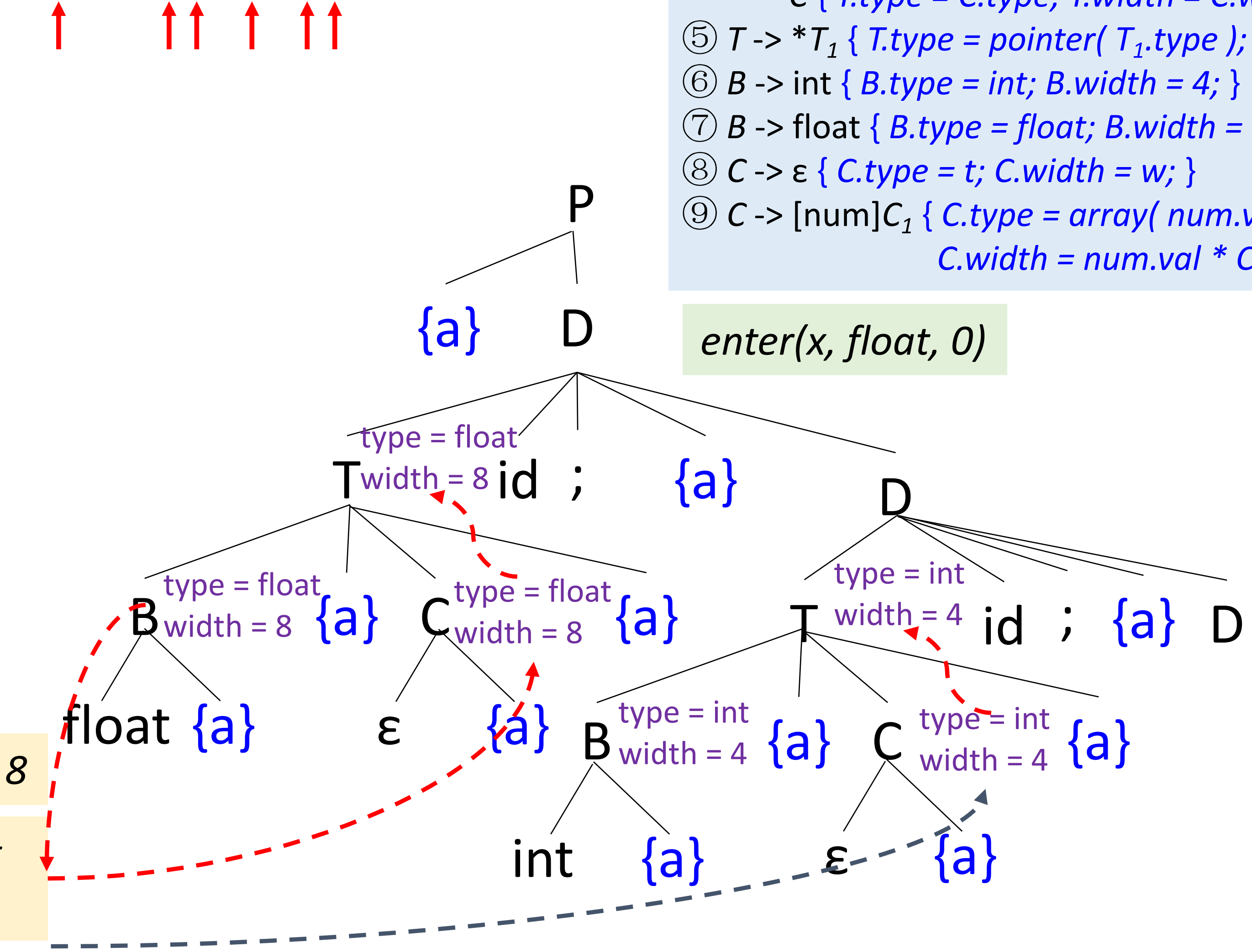


Example

- Input: float 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* -> float { *B.type = float; 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;* }

enter(x, float, 0)



offset = 8

t = float
w = 8

Example

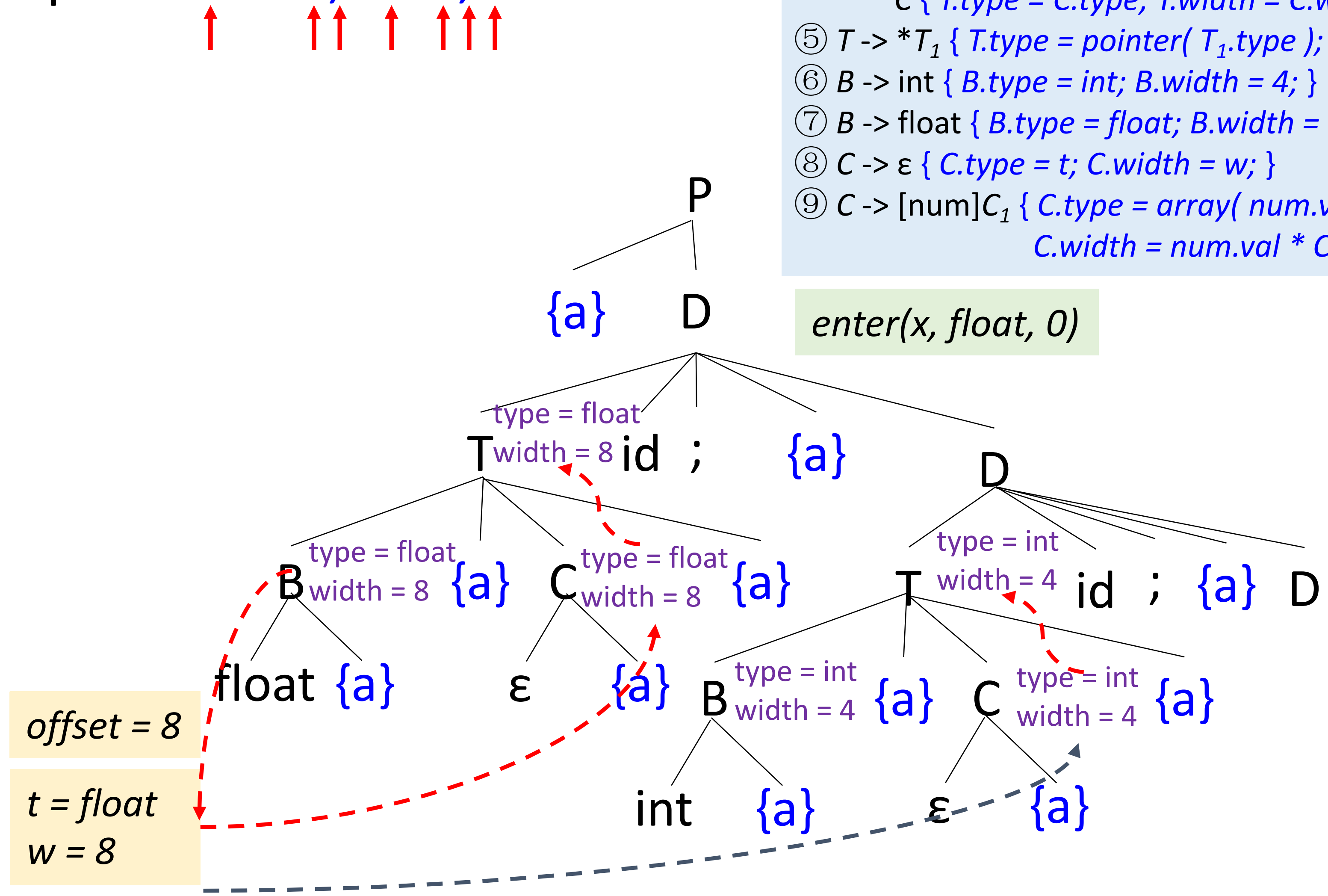
- Input: float x; int i;

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* -> float { *B.type = float; 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;* }

enter(x, float, 0)

offset = 8

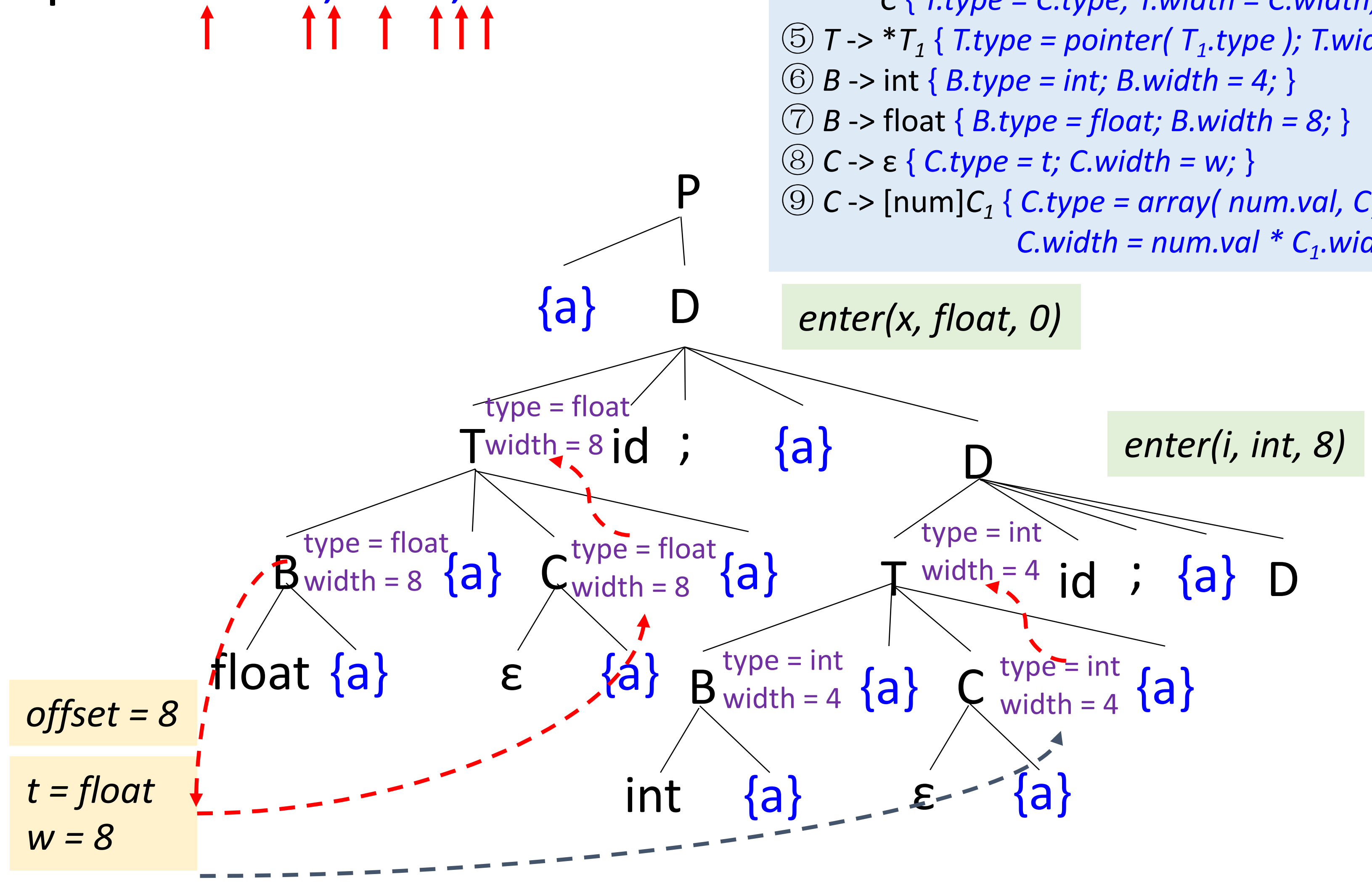
t = float
w = 8



Example

- Input: float x; int i;

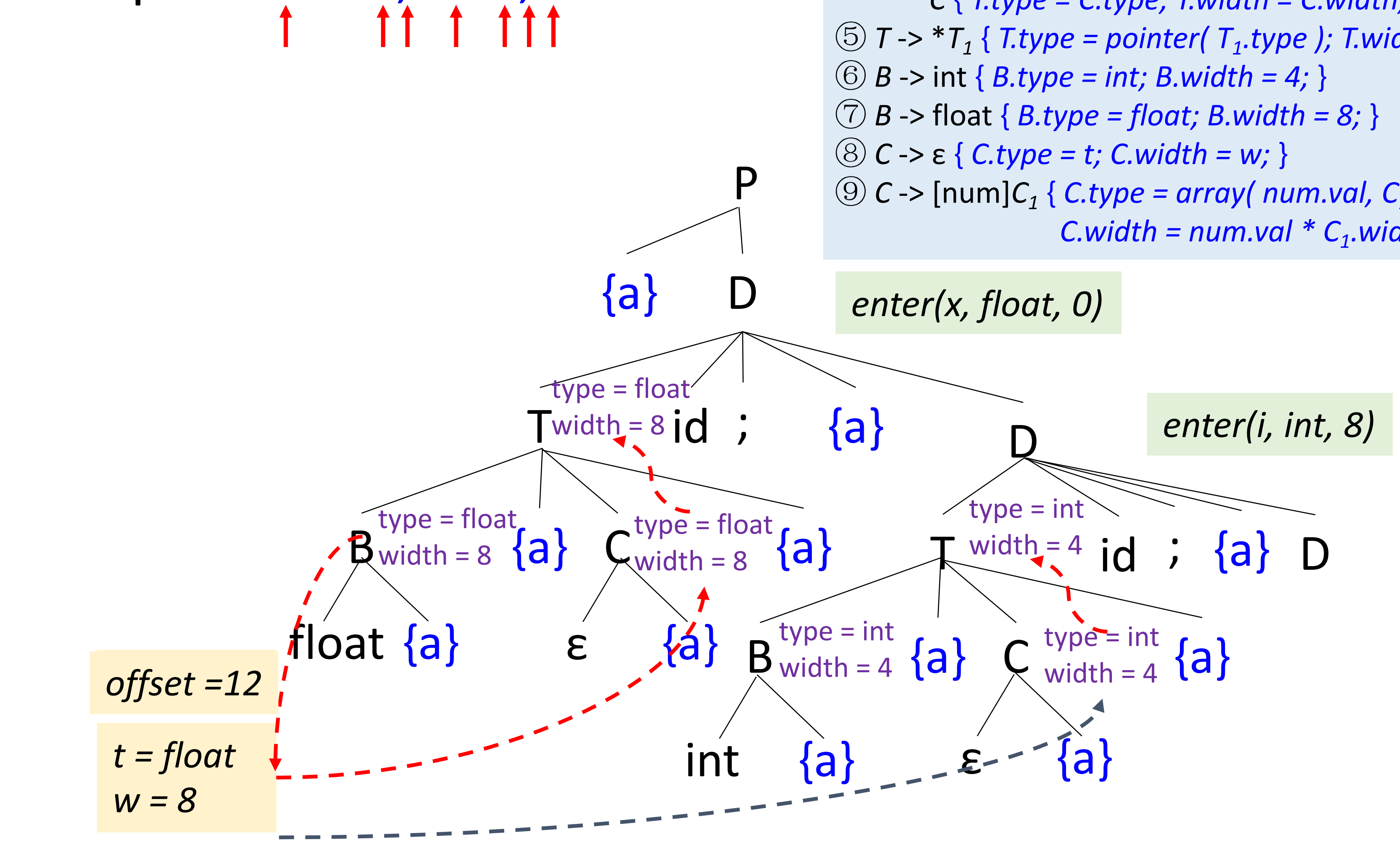
① *P* -> { *offset = 0* } *D*
② *D* -> *T* id; { *enter(id.lexeme, T.type, offset);*
offset = offset + T.width; } D_1
③ *D* -> ε
④ *T* -> *B* { *t = B.type; w = B.width;* }
C { *T.type = C.type; T.width = C.width;* }
⑤ *T* -> *T_1 { *T.type = pointer(T_1.type); T.width = 4;* }
⑥ *B* -> int { *B.type = int; B.width = 4;* }
⑦ *B* -> float { *B.type = float; 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: float x; int i;

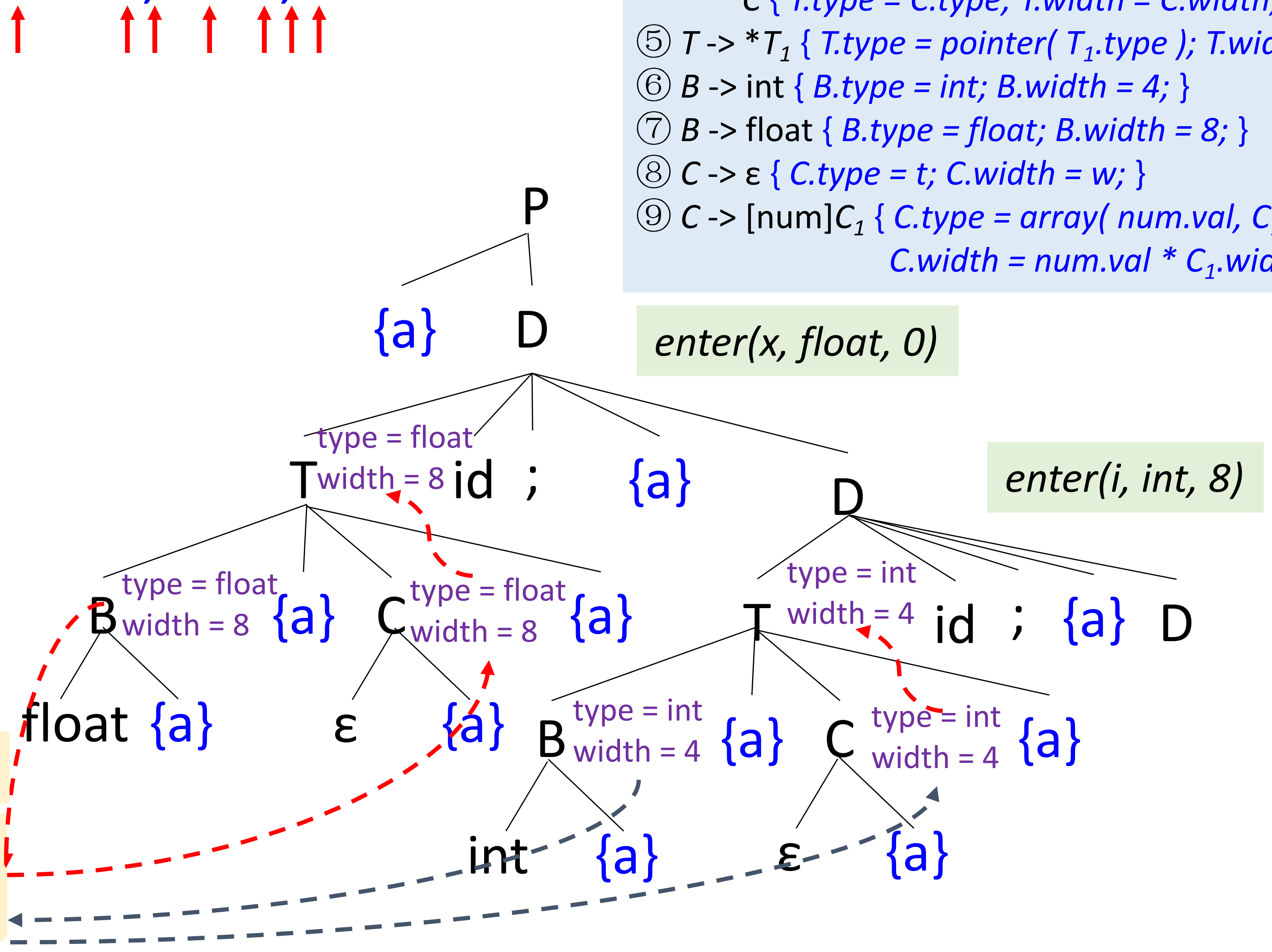
① $P \rightarrow$ { *offset = 0* } *D*
② $D \rightarrow T$ id; { *enter(id.lexeme, T.type, offset);*
offset = offset + T.width; } D_1
③ $D \rightarrow \varepsilon$
④ $T \rightarrow B$ { *t = B.type; w = B.width;* }
C { *T.type = C.type; T.width = C.width;* }
⑤ $T \rightarrow *T_1$ { *T.type = pointer(*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



Example

- Input: float 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 -> float { *B.type = float; 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

Code attributes can be long strings

```
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

Code attributes can be long strings

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

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

② $E \to E_1 + E_2$; { *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 $E \to (E_1)$ { *E.addr = E_1.addr;*

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

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

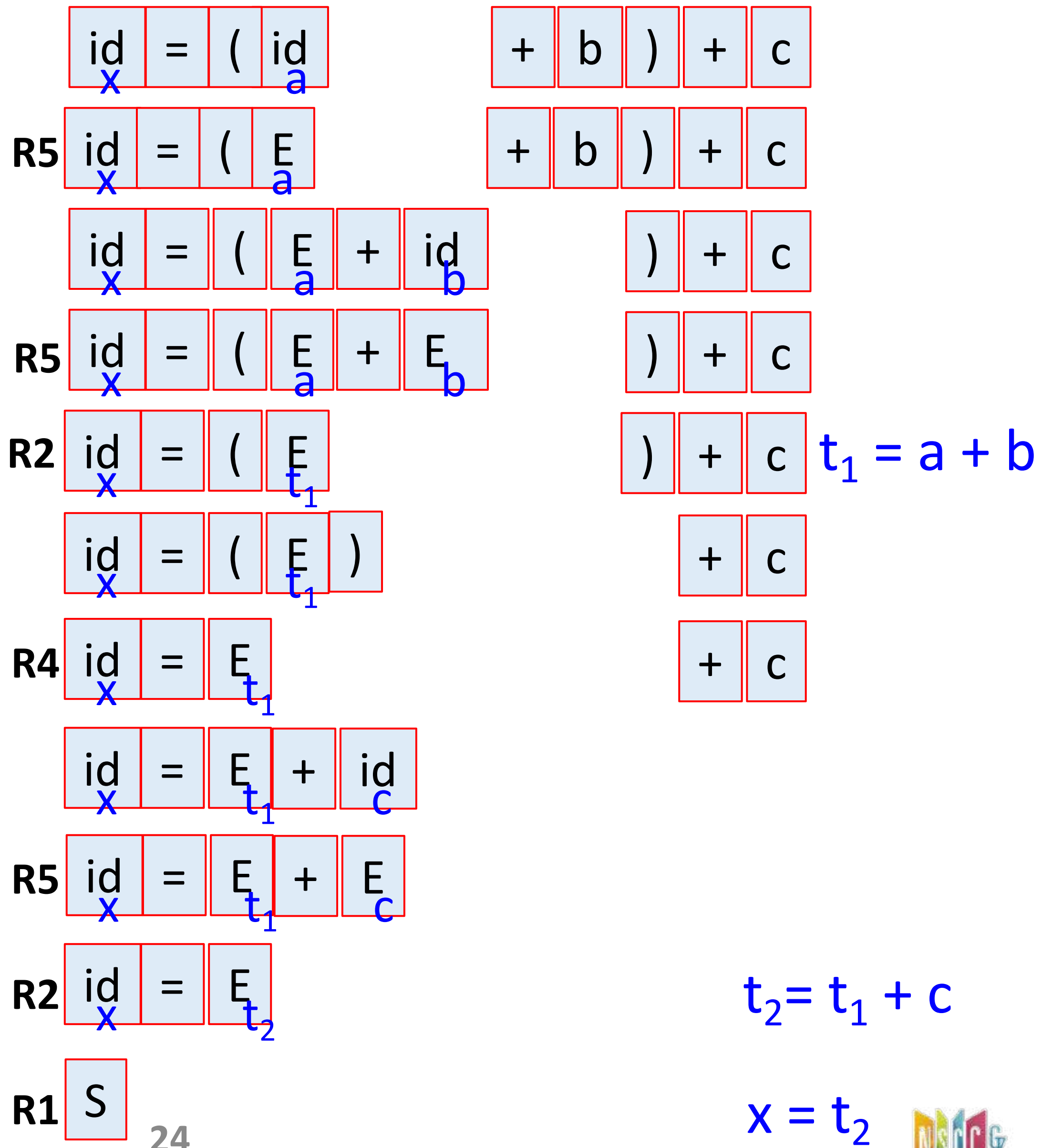
  $x = (a + b) + c$

- Translated TAC

  $t_1 = a + b$

  $t_2 = t_1 + c$

  $x = t_2$

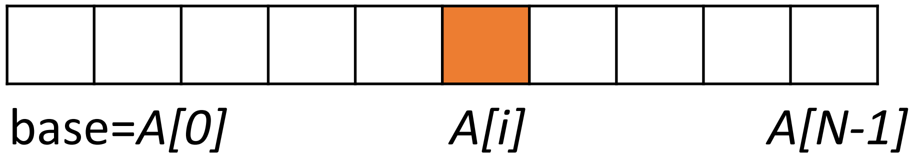


CodeGen: Array Reference[数组引用]

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

  int A[N];

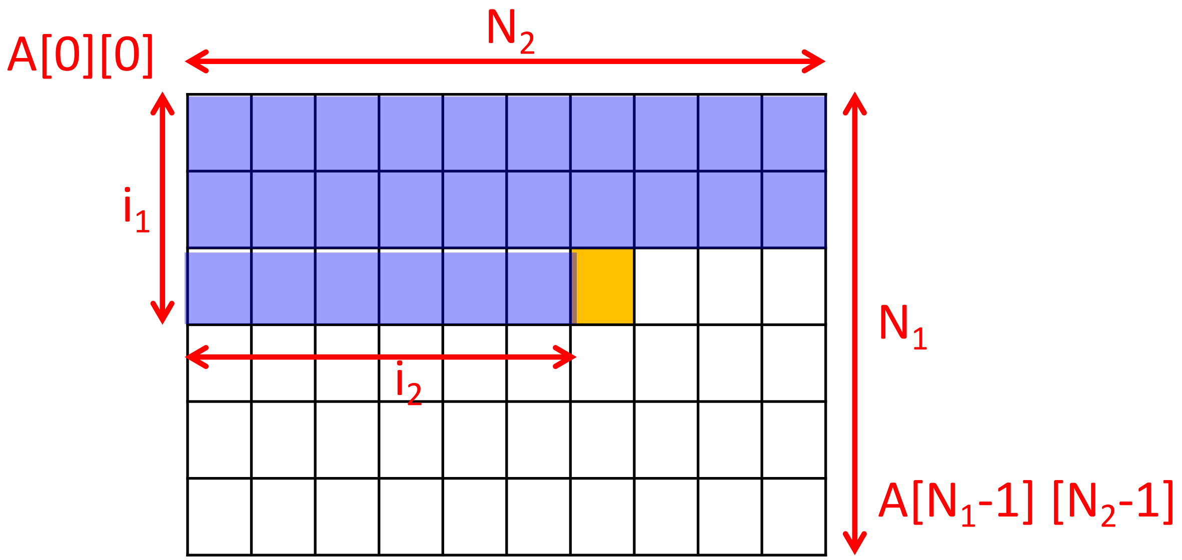
  A[i] ++;



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



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
  %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
}
```

// addr(@arr + 4x(0 + 2*3*4 + 3*4 + 4))

Builder.CreateInBoundsGEP(addr, …);

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

Example: LLVM

```c
double x;

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

    if (b < 5) b = 1;
    b = d < b;
}
```

```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
  %5 = load i32, i32* %2, align 4
  %6 = icmp slt i32 %5, 5                    // %6 = (b < 5)
  br i1 %6, label %7, label %8               // true: '7', false: '8'

7:                                                    ; preds = %0
  store i32 1, i32* %2, align 4              // b = 1
  br label %8                                // jump to '8'

8:                                                    ; preds = %7, %0
  %9 = load i32, i32* %4, align 4            // %9 = d
  %10 = load i32, i32* %2, align 4           // %10 = b
  %11 = icmp slt i32 %9, %10                 // %11 = d < b
  %12 = zext i1 %11 to i32                   // %12 = %11
  store i32 %12, i32* %2, align 4            // b = %12
  ret void
}
```

llvm::BasicBlock::Create(…);
Builder.CreateCondBr(…); // Create a conditional 'br Cond, TrueDest, FalseDest' instruction.
Builder.SetInsertPoint(…);

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 S_1

```
// t5=F, skip S1
if (!t5) goto S.next
S1.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

```
while (E) {
  // E.true
}
  // E.false
…
```

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

```
    if (a < b) goto E.true        E为真：只要a < b真
    goto L1                        a < b假：继续评估
L1: if (c < d) goto L2             a < b假、 c < d真：继续评估
    goto E.false                  E为假：a < b假，c < d假
L2: 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
```

SDT Translation of Booleans[布尔表达式]

- B -> B_1 || B_2
  - B_1*.true* is same as *B.true*, B_2 must be evaluated if B_1 is false[B_1假才评估B_2]
  - The true and false exits of B_2 are the same as *B*[B_2与B同真假]
- B -> E_1 *relop* E_2
  - Translated directly into a comparison TAC inst with jumps

B_1为真，跳转到B.true　　　　B_1为假，跳转到别处（需要继续评估B_2）

① *B* -> { B_1*.true = B.true;* B_1*.false = newlabel();* } B_1
　　|| { *label(*B_1*.false);* B_2*.true = B.true;* B_2*.false = B.false;* } B_2
② *B* -> { B_1*.true = newlabel();* B_1*.false = B.false;* } B_1
　　&& { *label(*B_1*.true);* B_2*.true = B.true;* B_2*.false = B.false;* } B_2
③ *B* -> E_1 *relop* E_2 { *gen('if'* E_1*.addr relop* E_2*.addr 'goto' B.true);*
　　gen('goto' B.false); }
④ *B* -> ! { B_1*.true = B.false;* B_1*.false = B.true;* } B_1
⑤ *B* -> true { *gen('goto' B.true);* }
⑥ *B* -> false { *gen('goto' B.false);* }

B: a boolean expression
S: a statement