

# Clang Tutorial

---

**Moonzoo Kim**  
**School of Computing**  
**KAIST**

*The original slides were written by Yongbae Park, [yongbae2@gmail.com](mailto:yongbae2@gmail.com)*

# Content

- Motivation of learning code analysis technique
- Overview of Clang
- AST structure of Clang
  - Decl class
  - Stmt class
- Traversing Clang AST

# Motivation for Learning Code Analysis Technique

- Biologists know how to analyze laboratory mice. In addition, they know how to modify the mice by applying new medicine or artificial organ
- Mechanical engineers know how to analyze and modify mechanical products using CAD tools.
- Software engineers also have to know how to analyze and modify software code which is far more complex than any engineering product. Thus, software analysis/modification requires automated analysis tools.
  - Using source level analysis framework (e.g., Clang, C Intermediate Language (CIL), EDG parser)
  - Using low-level intermediate representation (IR) analysis framework (e.g., LLVM IR)

# Overview

- There are frequent chances to analyze/modify program code mechanically/automatically
  - Ex1. Refactoring code for various purposes
  - Ex2. Generate test driver automatically
  - Ex3. Insert probes to monitor target program behavior
- Clang is a library to convert a C program into an abstract syntax tree (AST) and manipulate the AST
  - Clang, the LLVM C/C++ front-end supports the full-features of C/C++ and compatible with GCC
  - Ex) finding branches, renaming variables, pointer alias analysis, etc
- **Clang is particularly useful to simply modify C/C++ code**
  - Ex1. Add `printf("Branch Id:%d\n", bid)` at each branch
  - Ex2. Add `assert(pt != null)` right before referencing `pt`

# Example C code

- 2 functions are declared:  
myPrint and main
  - main function calls myPrint and returns 0
  - myPrint function calls printf
    - myPrint contains if and for statements
- 1 global variable is declared:  
global

```
//clang_example.c
#include <stdio.h>

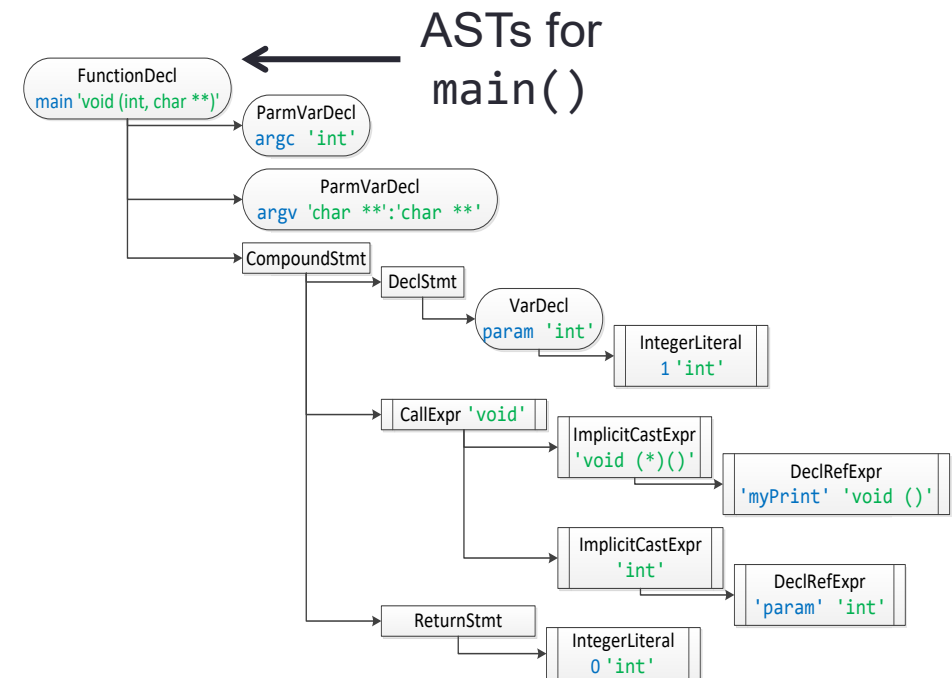
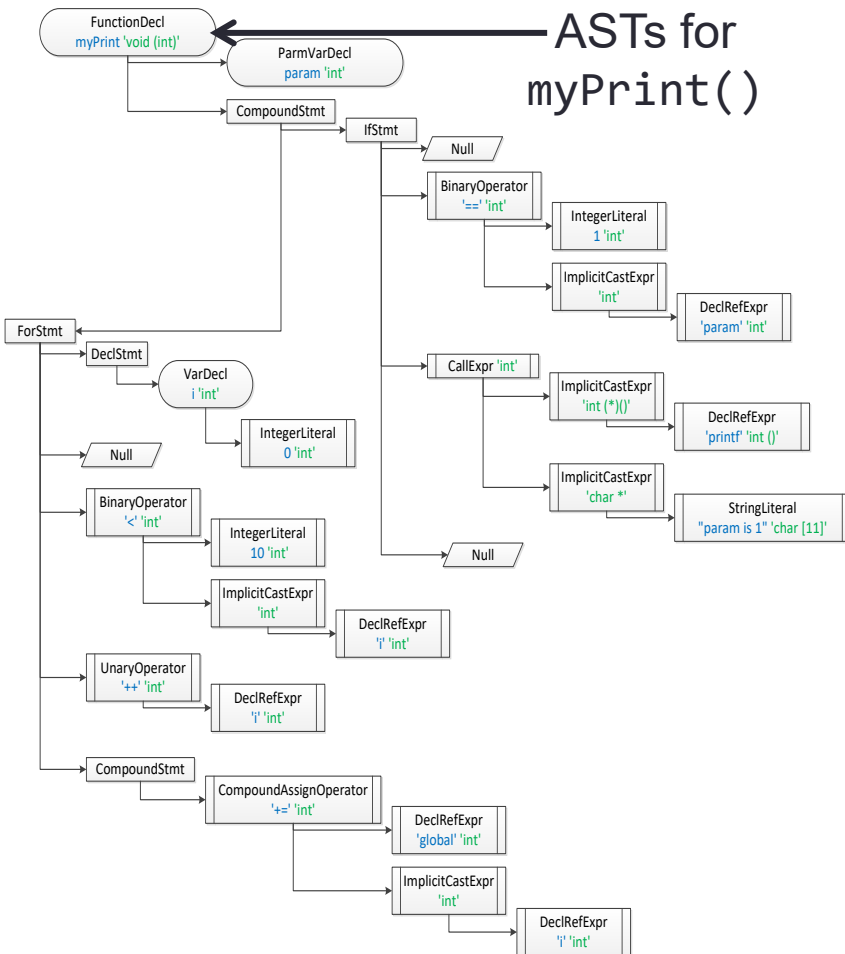
int global;

void myPrint(int param) {
    if (param == 1)
        printf("param is 1");
    for (int i = 0 ; i < 10 ; i++ ) {
        global += i;
    }
}

int main(int argc, char *argv[]) {
    int param = 1;
    myPrint(param);
    return 0;
}
```

# Example AST

- Clang generates 3 ASTs for `myPrint()`, `main()`, and global
  - A function declaration has a function body and parameters



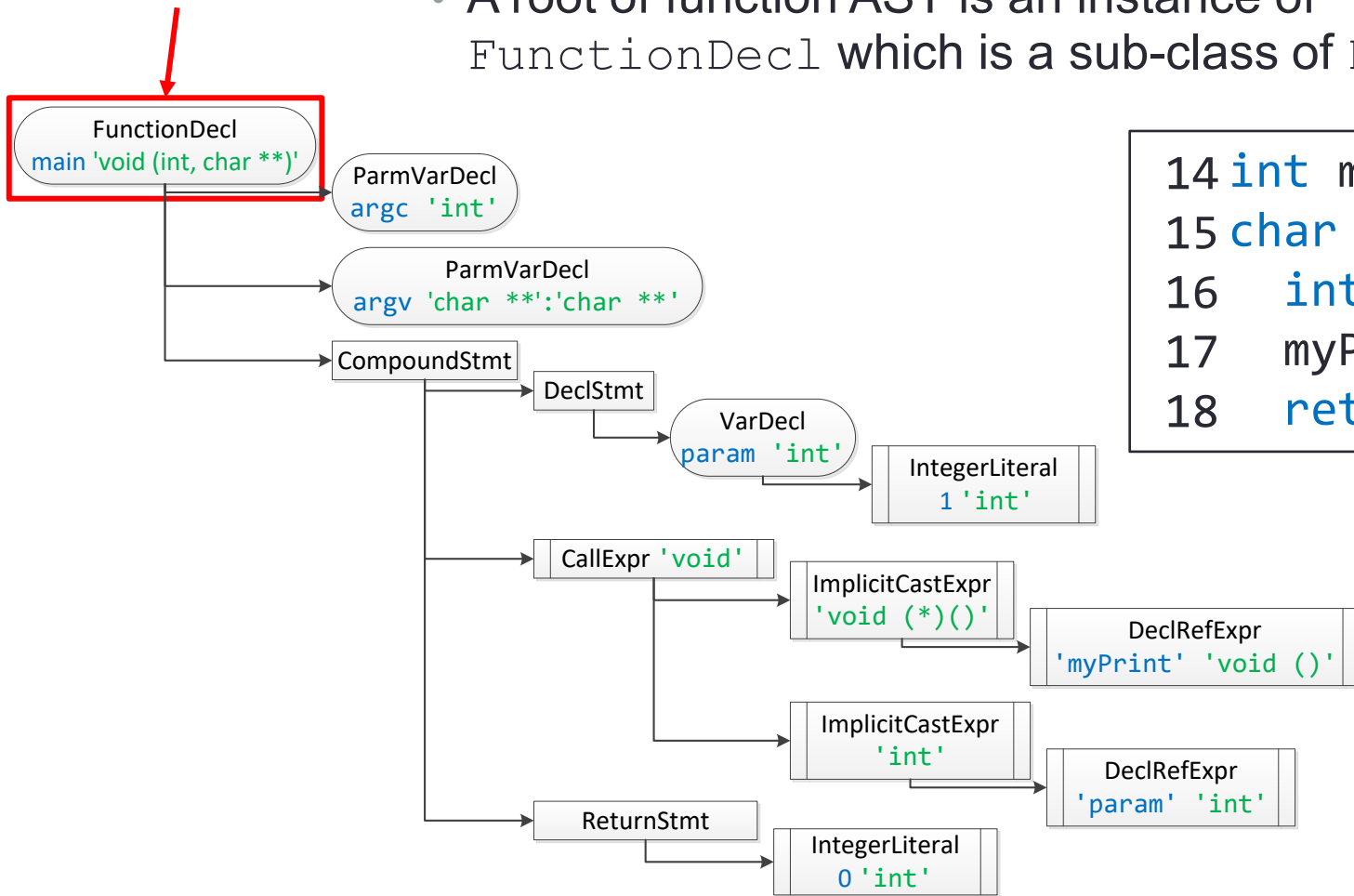
# Structure of AST

- Each node in AST is an instance of either `Decl` or `Stmt` class
  - `Decl` represents declarations and there are sub-classes of `Decl` for different declaration types
    - Ex) `FunctionDecl` class for function declaration and `ParmVarDecl` class for function parameter declaration
  - `Stmt` represents statements and there are sub-classes of `Stmt` for different statement types
    - Ex) `IfStmt` for `if` and `ReturnStmt` class for function return
- Comments (i.e. `/* */`, `//`) are not built into an AST

# Decl (1/4)

- A root of the function AST is a Decl node
- A root of function AST is an instance of FunctionDecl which is a sub-class of Decl

Function declaration



```

14 int main(int argc,
15 char *argv[]) {
16     int param = 1;
17     myPrint(param);
18     return 0;}
  
```

## Legend

Declaration type  
name type

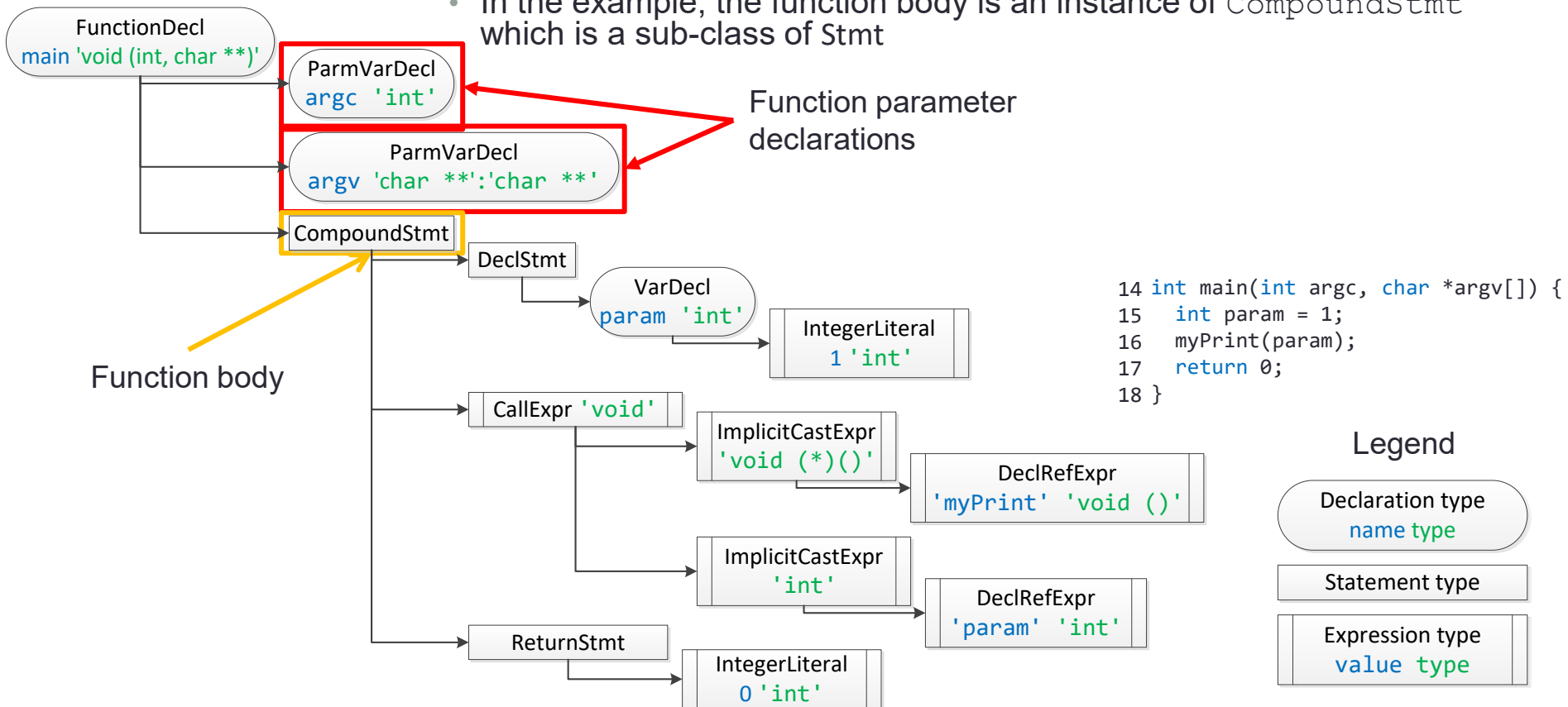
Statement type

Expression type  
value type



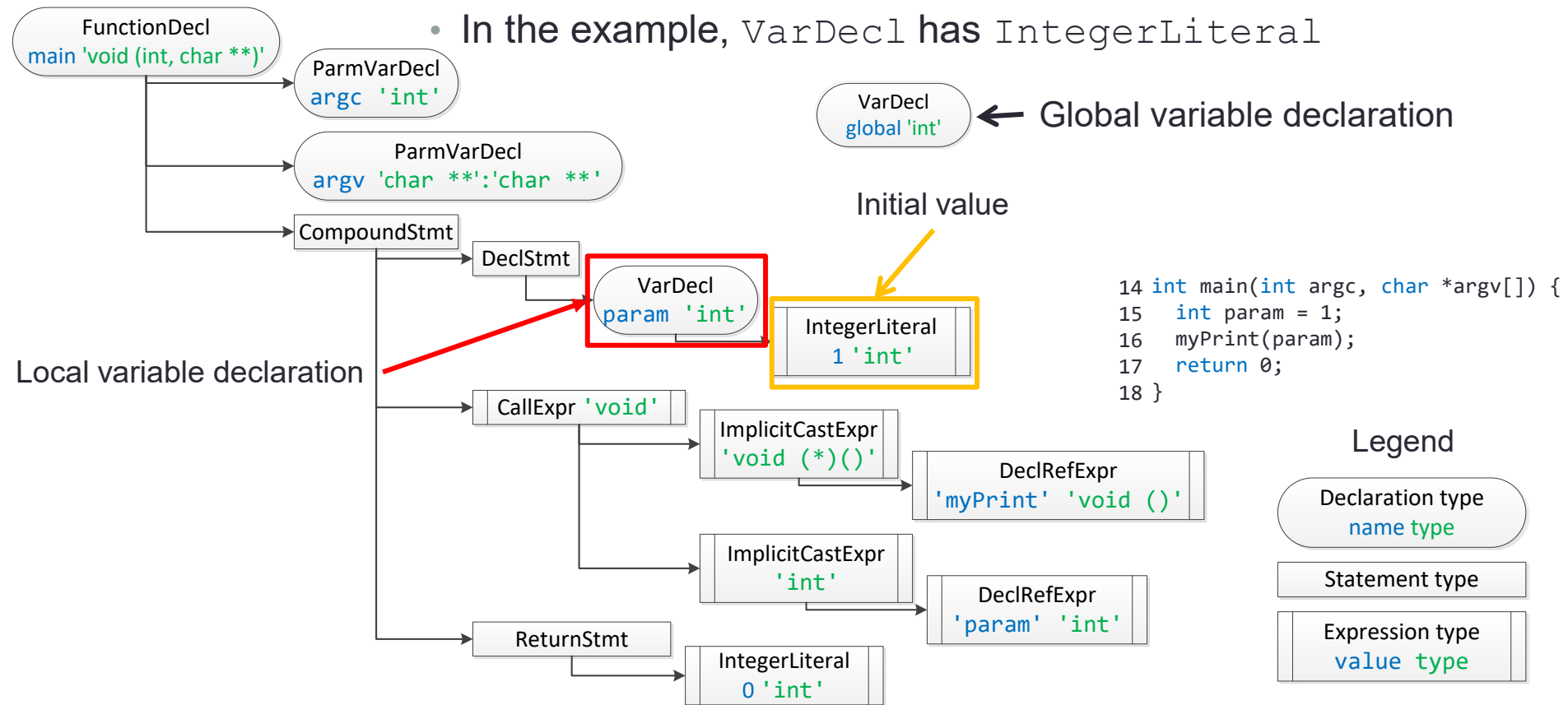
# Decl (2/4)

- FunctionDecl can have an instance of ParmVarDecl for a function parameter and a function body
  - ParmVarDecl is a child class of Decl
  - Function body is an instance of Stmt
    - In the example, the function body is an instance of CompoundStmt which is a sub-class of Stmt



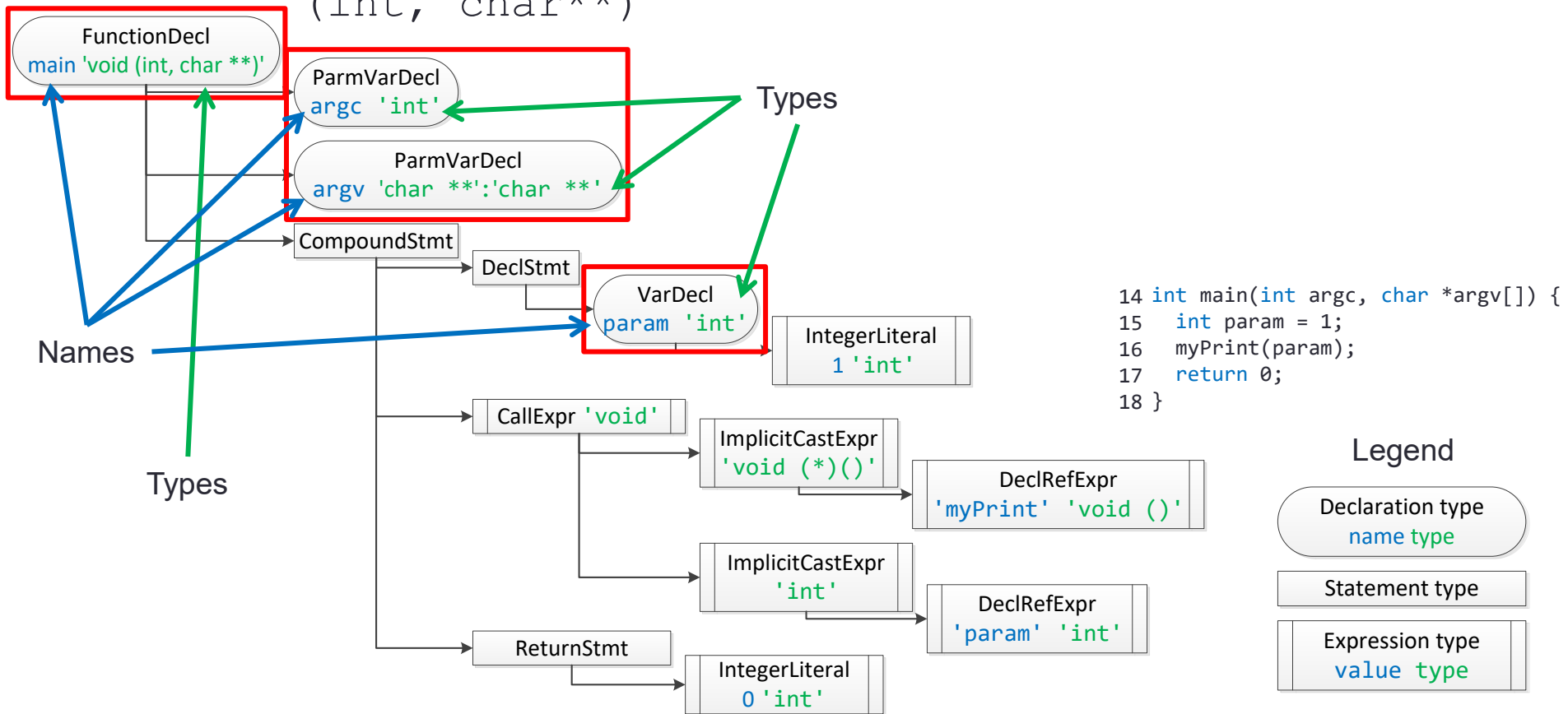
# Decl (3/4)

- VarDecl is for a local and global variable declaration
  - VarDecl has a child if a variable has a initial value
    - In the example, VarDecl has IntegerLiteral



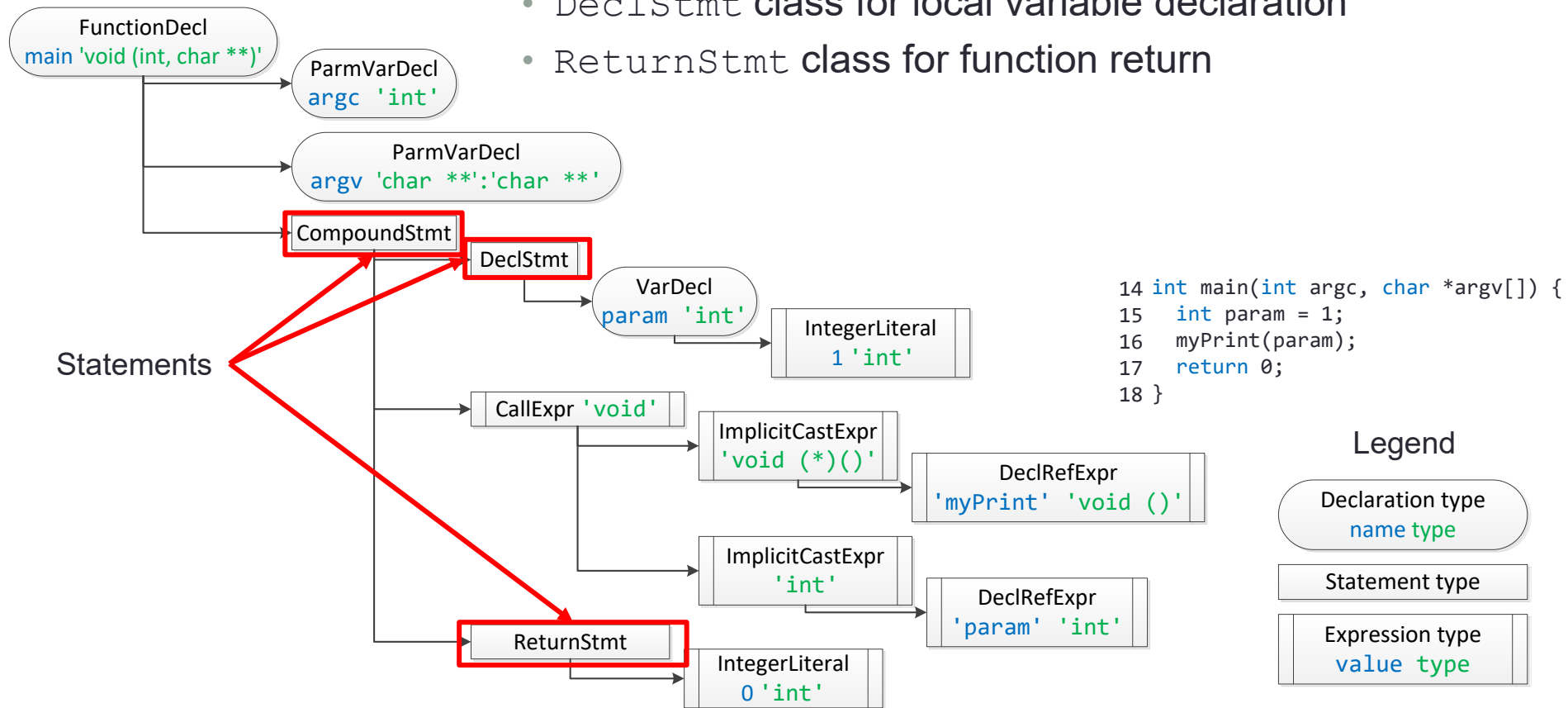
# Decl (4/4)

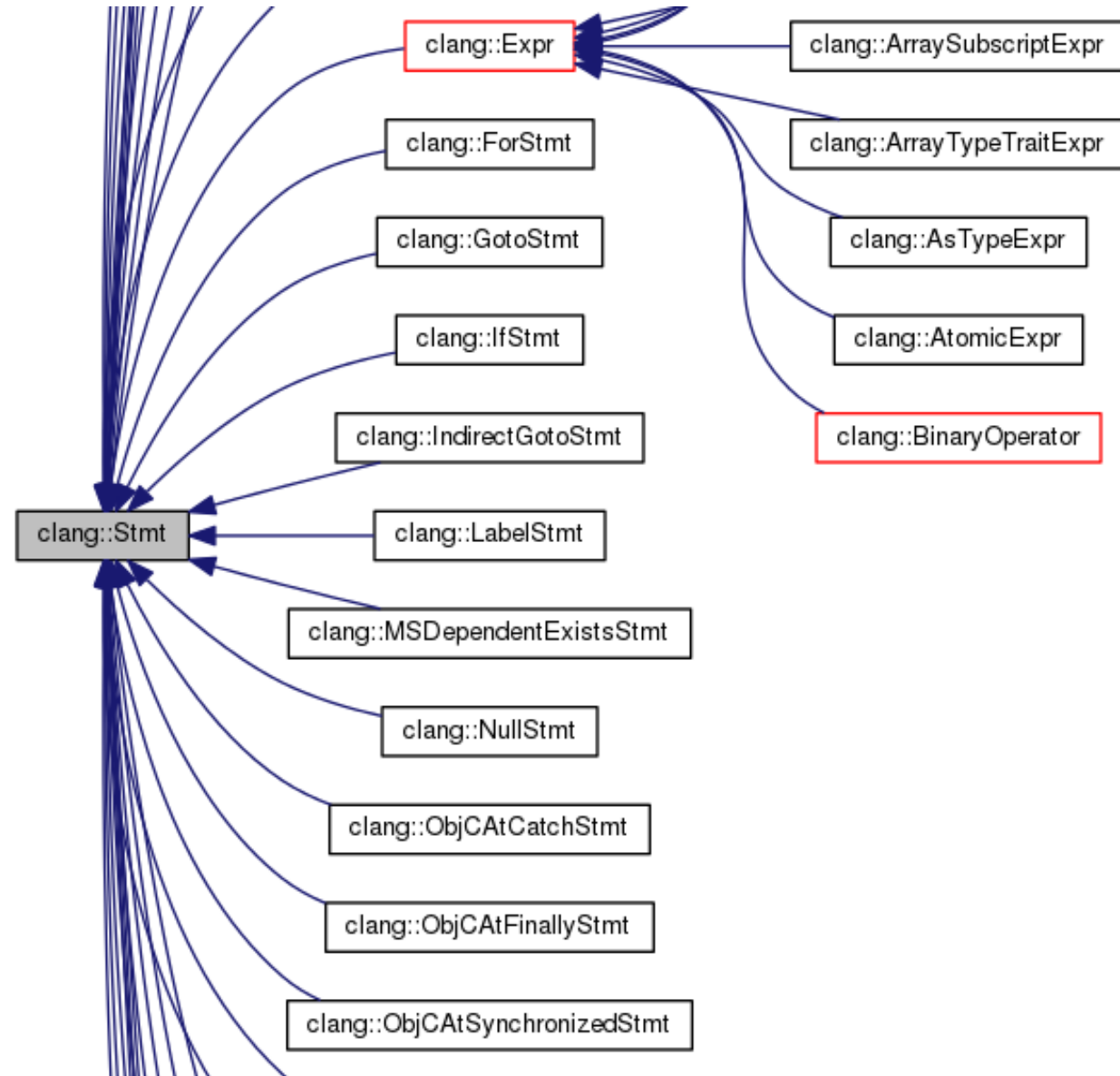
- FunctionDecl, ParmVarDecl and VarDecl have a name and a type of declaration
  - Ex) FunctionDecl has a name 'main' and a type 'void (int, char\*\*)'



# Stmt (1/9)

- Stmt represents a statement
  - Subclasses of Stmt
    - CompoundStmt class for code block
    - DeclStmt class for local variable declaration
    - ReturnStmt class for function return

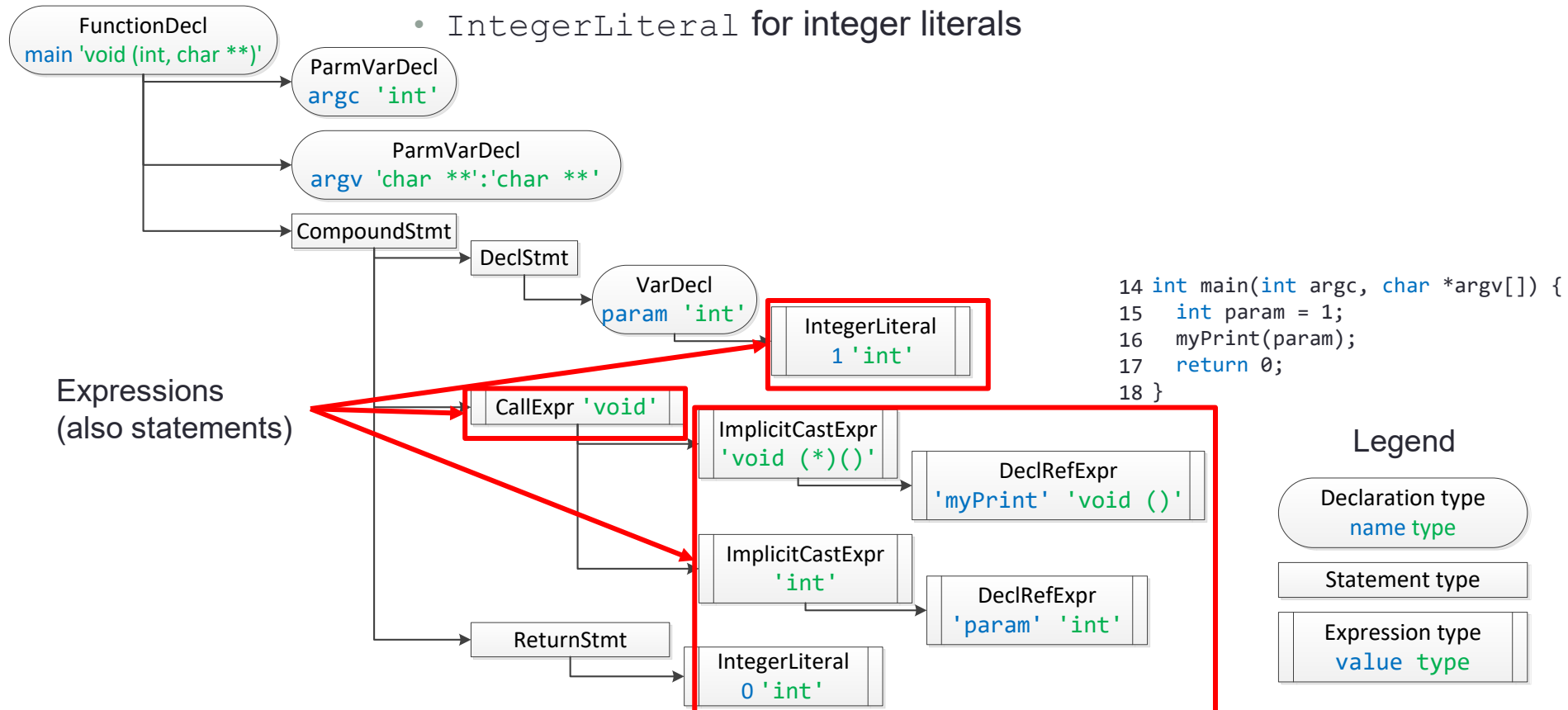




<https://clang.llvm.org/doxygen/>

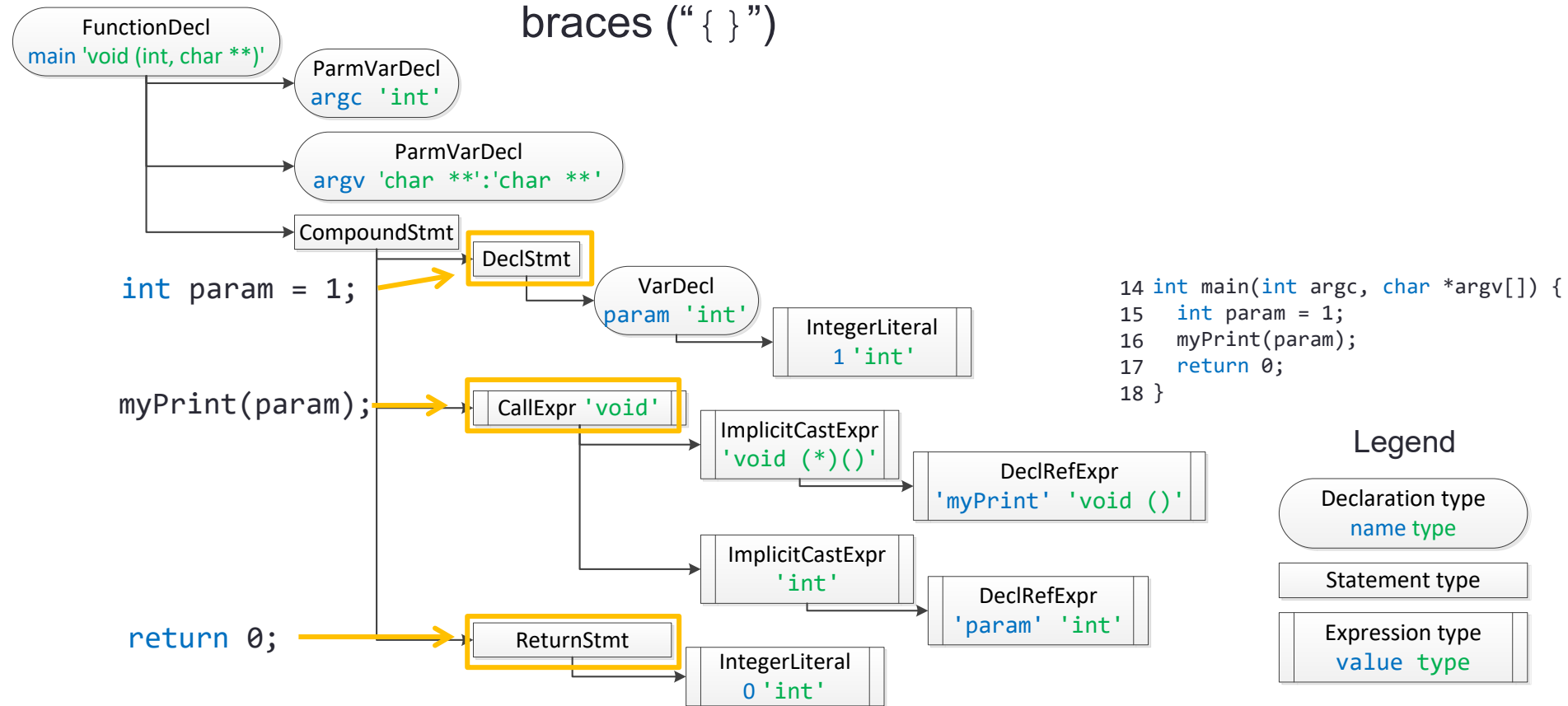
# Stmt (2/9)

- Expr represents an expression (a subclass of Stmt)
  - Subclasses of Expr
    - CallExpr for function call
    - ImplicitCastExpr for implicit type casts
    - DeclRefExpr for referencing declared variables and functions
    - IntegerLiteral for integer literals



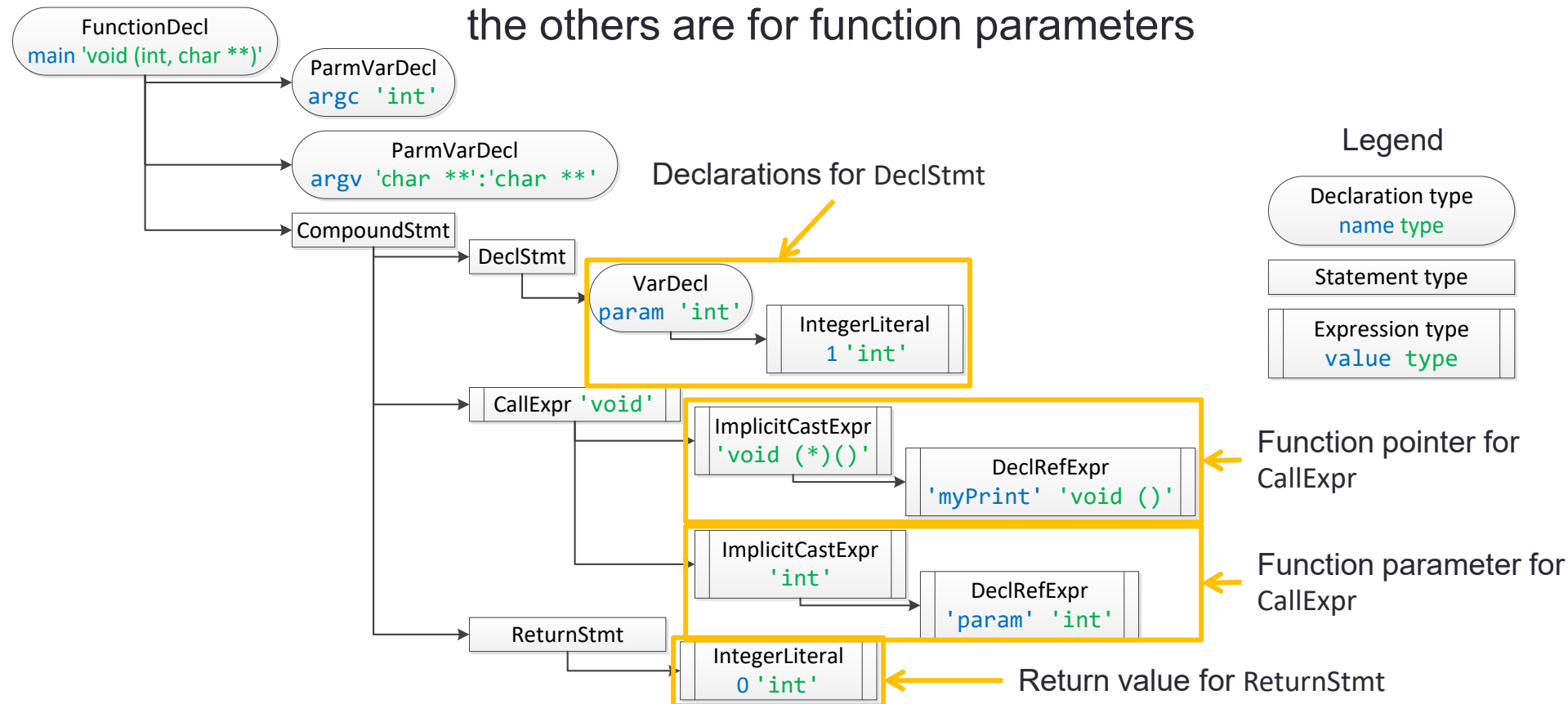
# Stmt (3/9)

- Stmt may have a child containing additional information
  - CompoundStmt has statements in a code block of braces (“{ }”)



# Stmt (4/9)

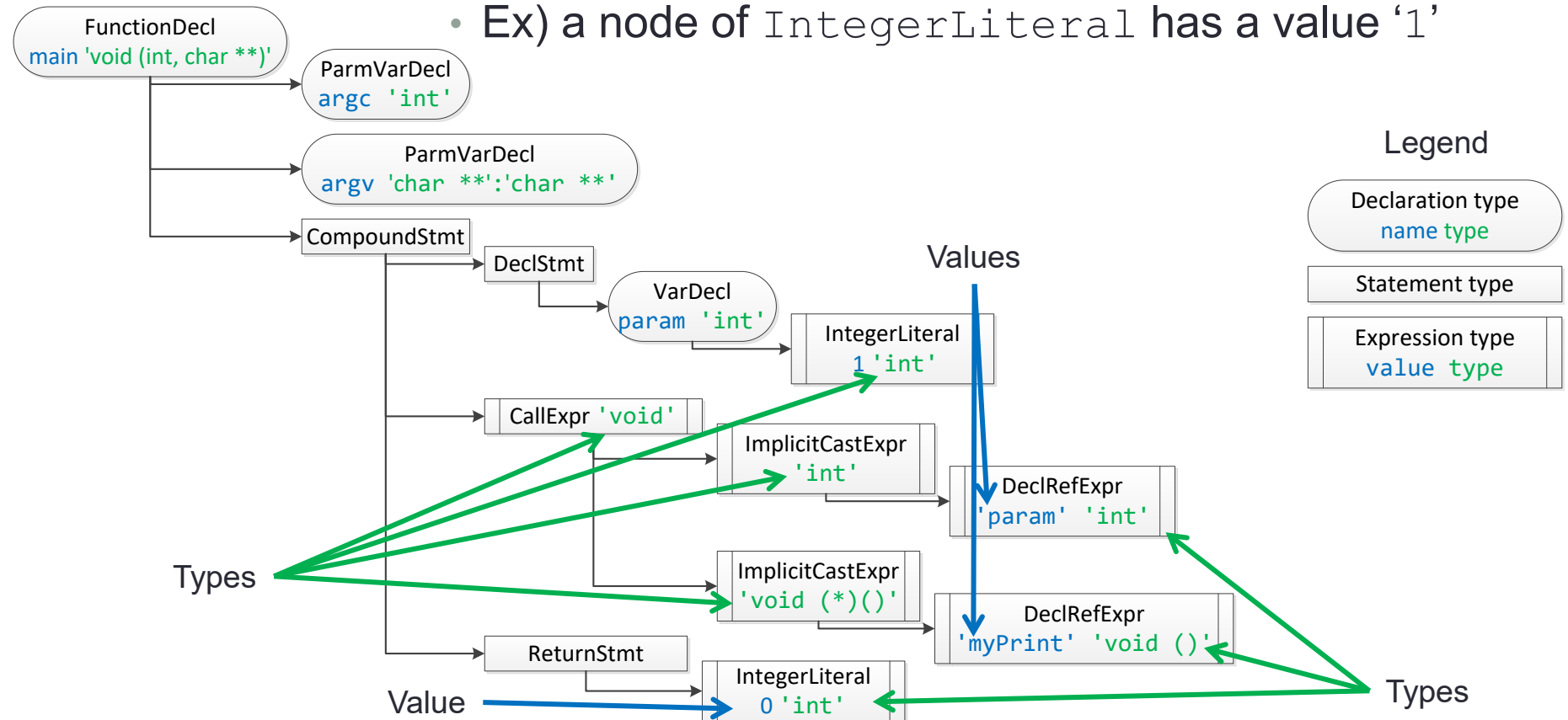
- Stmt may have a child containing additional information (cont')
  - The first child of CallExpr is for a function pointer and the others are for function parameters





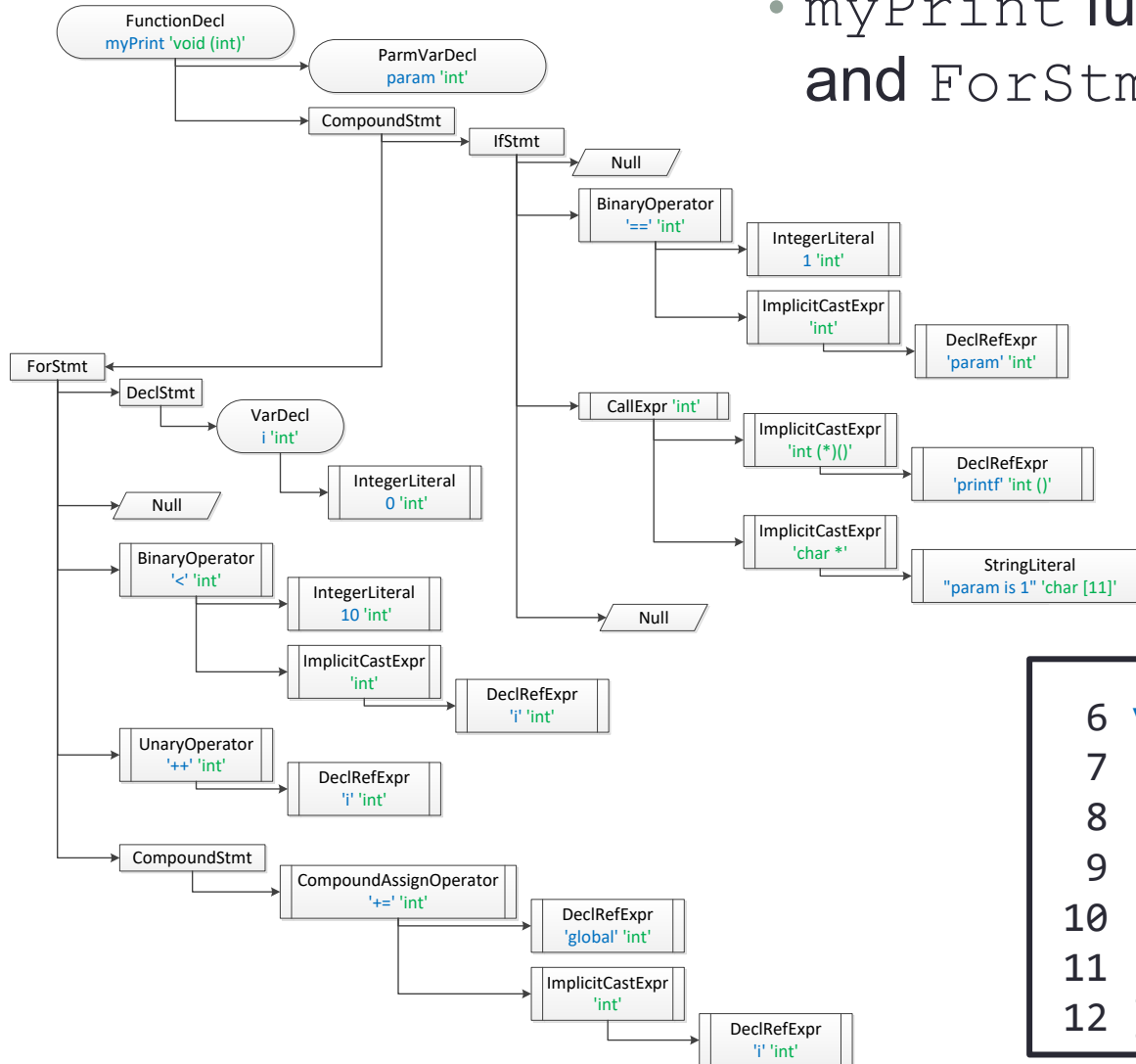
# Stmt (5/9)

- Expr has a type of an expression
  - Ex) a node of CallExpr has a type 'void'
- Some sub-classes of Expr can have a value
  - Ex) a node of IntegerLiteral has a value '1'



# Stmt (6/9)

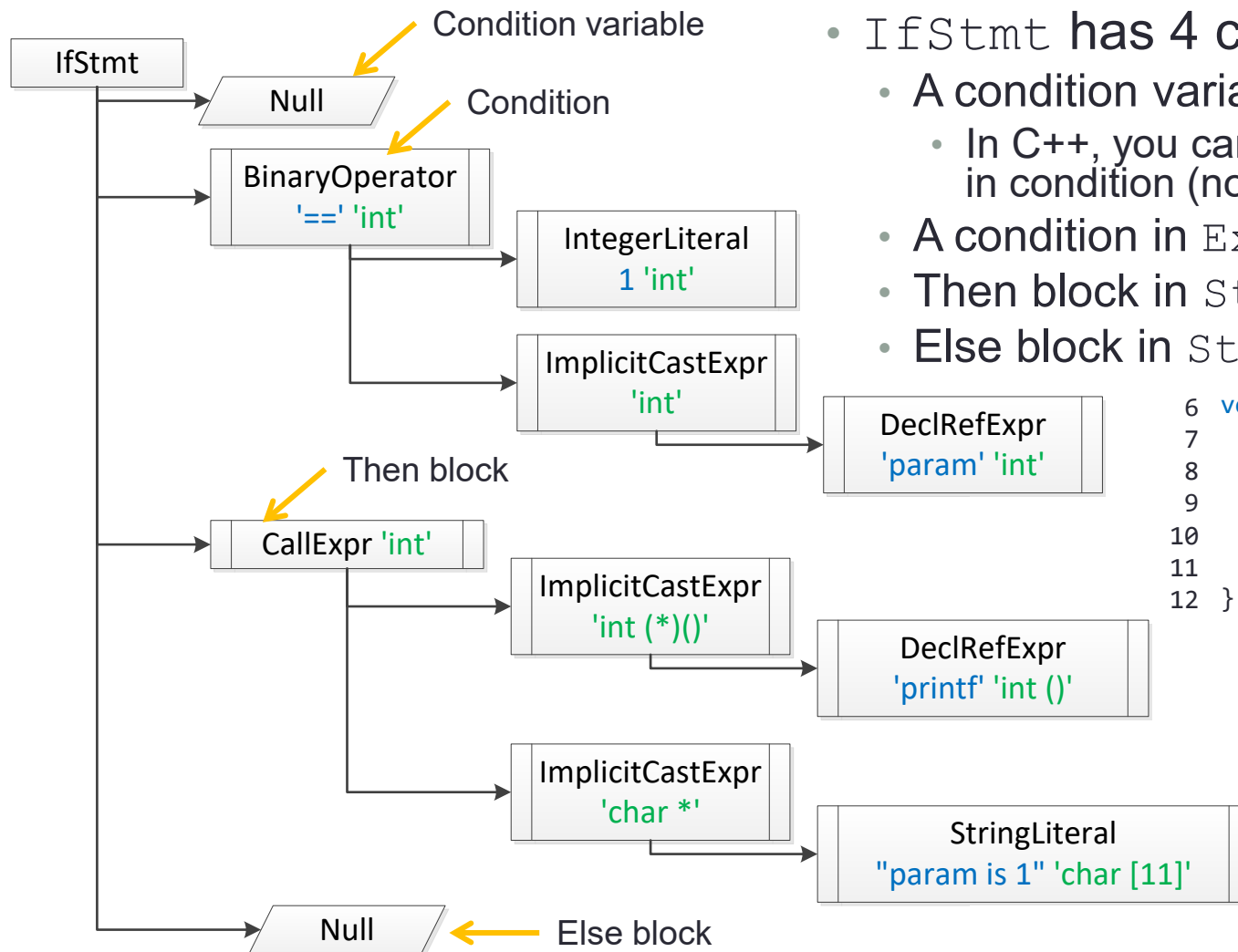
- myPrint function contains IfStmt and ForStmt in its function body



```

6 void myPrint(int param) {
7   if (param == 1)
8     printf("param is 1");
9   for (int i=0;i<10;i++) {
10    global += i;
11  }
12 }
  
```

# Stmt (7/9)



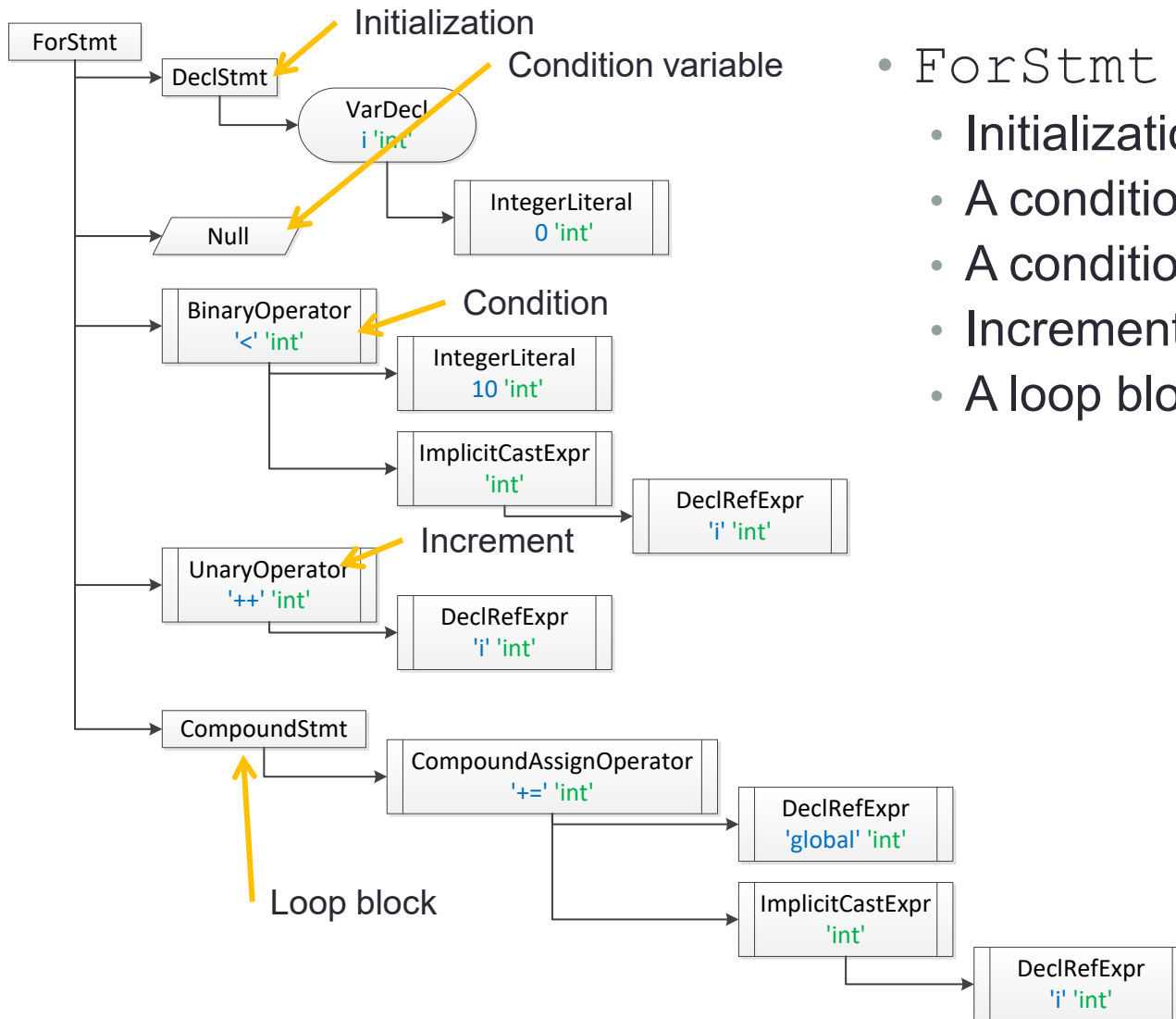
- `IfStmt` has 4 children

- A condition variable in `VarDecl`
  - In C++, you can declare a variable in condition (not in C)
- A condition in `Expr`
- Then block in `Stmt`
- Else block in `Stmt`

```

6 void myPrint(int param) {
7   if (param == 1)
8     printf("param is 1");
9   for (int i = 0 ; i < 10 ; i++ ) {
10    global += i;
11  }
12 }
  
```

# Stmt (8/9)



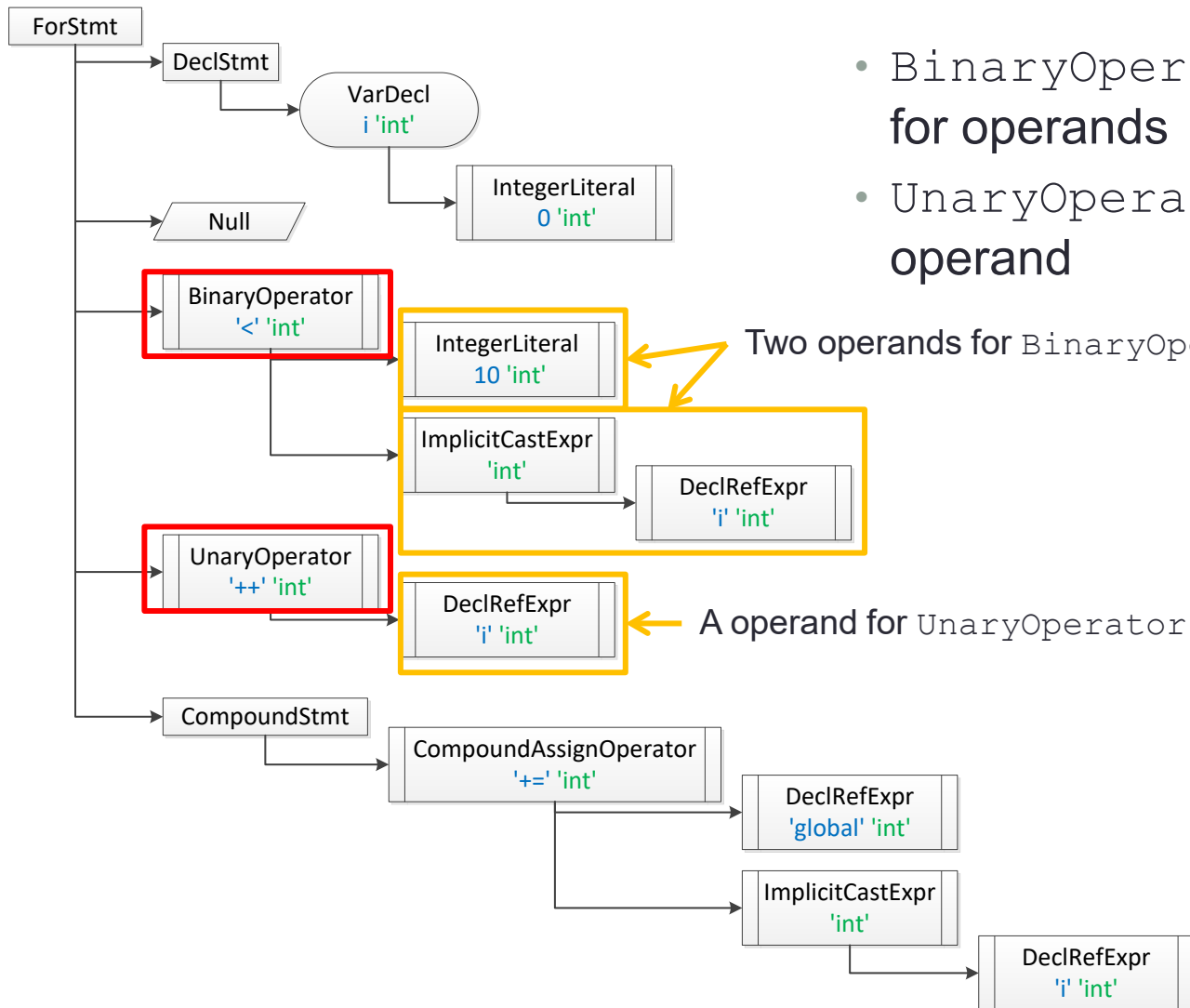
- ForStmt has 5 children

- Initialization in Stmt
- A condition variable in VarDecl
- A condition in Expr
- Increment in Expr
- A loop block in Stmt

```

6 void myPrint(int param) {
7   if (param == 1)
8     printf("param is 1");
9   for (int i = 0 ; i < 10 ; i++ ) {
10    global += i;
11  }
12 }
  
```

# Stmt (9/9)



- BinaryOperator has 2 children for operands
- UnaryOperator has a child for operand

```

6 void myPrint(int param) {
7   if (param == 1)
8     printf("param is 1");
9   for (int i = 0 ; i < 10 ; i++ ) {
10    global += i;
11  }
12 }
  
```

# Traversing Clang AST (1/3)

- Clang provides a visitor design pattern for user to access AST
- ParseAST() starts building and traversal of an AST:

`void clang::ParseAST (Preprocessor &pp, ASTConsumer *C, ASTContext &Ctx, ...)`

- The callback function `HandleTopLevelDecl()` in `ASTConsumer` is called for each top-level declaration
  - `HandleTopLevelDecl()` receives a list of function and global variable declarations as a parameter
- A user has to customize `ASTConsumer` to build his/her own program analyzer

```
1 class MyASTConsumer : public ASTConsumer
2 {
3     public:
4     MyASTConsumer(Rewriter &R) {}
5
6     virtual bool HandleTopLevelDecl(DeclGroupRef DR) {
7         for(DeclGroupRef::iterator b=DR.begin(), e=DR.end(); b!=e;++b){
8             ... // variable b has each decleration in DR
9         }
10        return true;
11    }
12 };
```

# Traversing Clang AST (2/3)

- HandleTopLevelDecl() calls TraverseDecl() which recursively travel a target AST from the top-level declaration by calling VisitStmt (), VisitFunctionDecl(), etc.

```

1 class MyASTVisitor : public RecursiveASTVisitor<MyASTVisitor> {
2   bool VisitStmt(Stmt *s) { ← VisitStmt is called when Stmt is encountered
3     printf("\t%s \n", s->getStmtClassName() );
4     return true;
5   }
6   bool VisitFunctionDecl(FunctionDecl *f) { ← VisitFunctionDecl is called when
7     if (f->hasBody()) {                               FunctionDecl is encountered
8       Stmt *FuncBody = f->getBody();
9       printf("%s\n", f->getName());
10    }
11    return true;
12  }
13 };
14 class MyASTConsumer : public ASTConsumer {
15   virtual bool HandleTopLevelDecl(DeclGroupRef DR) {
16     for (DeclGroupRef::iterator b = DR.begin(), e = DR.end(); b != e; ++b) {
17       MyASTVisitor Visitor;
18       Visitor.TraverseDecl(*b);
19     }
20     return true;
21   }
22   ...
23 };

```

# Traversing Clang AST (3/3)

- VisitStmt() in RecursiveASTVisitor is called for every Stmt object in the AST. RecursiveASTVisitor visits each Stmt in a depth-first search order
  - If the return value of VisitStmt is false, recursive traversal halts
  - Example: main function of the previous example

