Clang Tutorial

Moonzoo Kim
School of Computing
KAIST

The original slides were written by Yongbae Park, 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
  - 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`

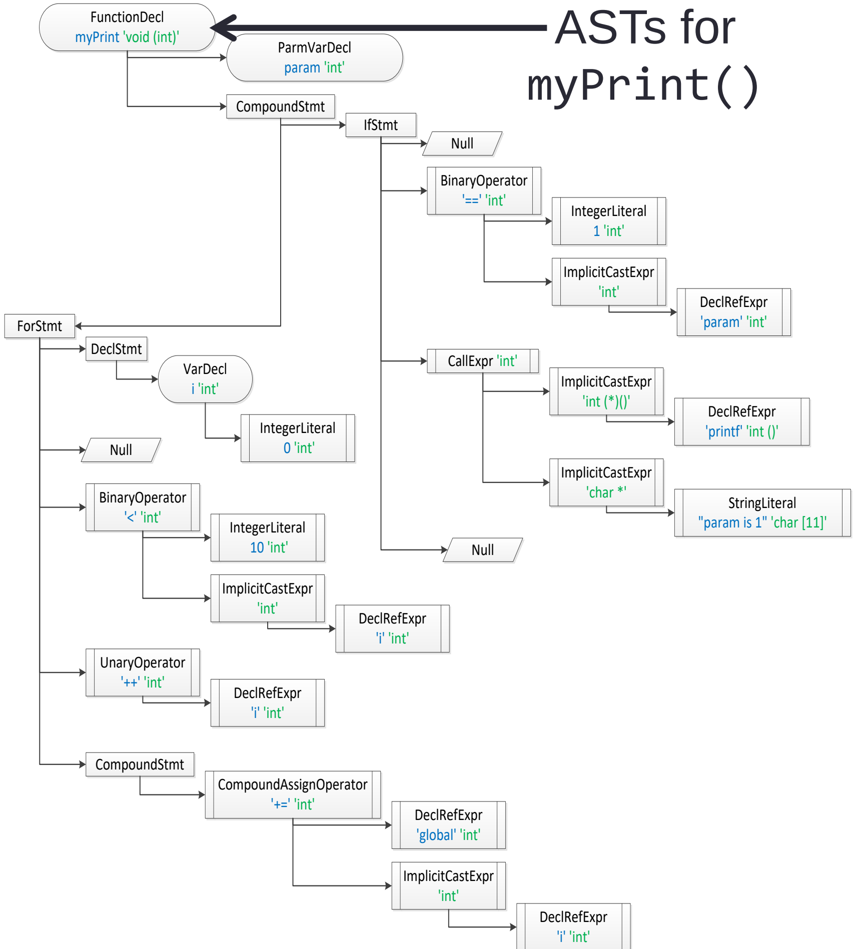
Example C code

- 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

```c
//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;
}
```

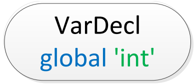
Example AST

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



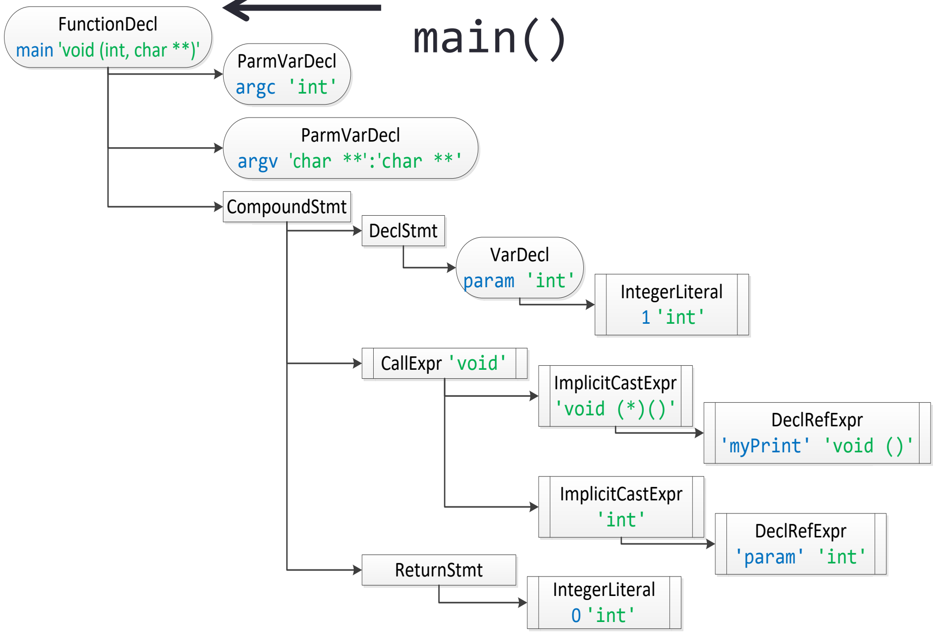
ASTs for `myPrint()`

FunctionDecl myPrint 'void (int)'
ParmVarDecl param 'int'
CompoundStmt
IfStmt
Null
BinaryOperator '==' 'int'
IntegerLiteral 1 'int'
ImplicitCastExpr 'int'
DeclRefExpr 'param' 'int'
CallExpr 'int'
ImplicitCastExpr 'int (*)()'
DeclRefExpr 'printf' 'int ()'
ImplicitCastExpr 'char *'
StringLiteral "param is 1" 'char [11]'
Null
ForStmt
DeclStmt
VarDecl i 'int'
IntegerLiteral 0 'int'
Null
BinaryOperator '<' 'int'
IntegerLiteral 10 'int'
ImplicitCastExpr 'int'
DeclRefExpr 'i' 'int'
UnaryOperator '++' 'int'
DeclRefExpr 'i' 'int'
CompoundStmt
CompoundAssignOperator '+=' 'int'
DeclRefExpr 'global' 'int'
ImplicitCastExpr 'int'
DeclRefExpr 'i' 'int'



VarDecl
global 'int'

AST for global



ASTs for `main()`

FunctionDecl main 'void (int, char **)'
ParmVarDecl argc 'int'
ParmVarDecl argv 'char **':'char **'
CompoundStmt
DeclStmt
VarDecl param 'int'
IntegerLiteral 1 'int'
CallExpr 'void'
ImplicitCastExpr 'void (*)()'
DeclRefExpr 'myPrint' 'void ()'
ImplicitCastExpr 'int'
DeclRefExpr 'param' 'int'
ReturnStmt
IntegerLiteral 0 'int'

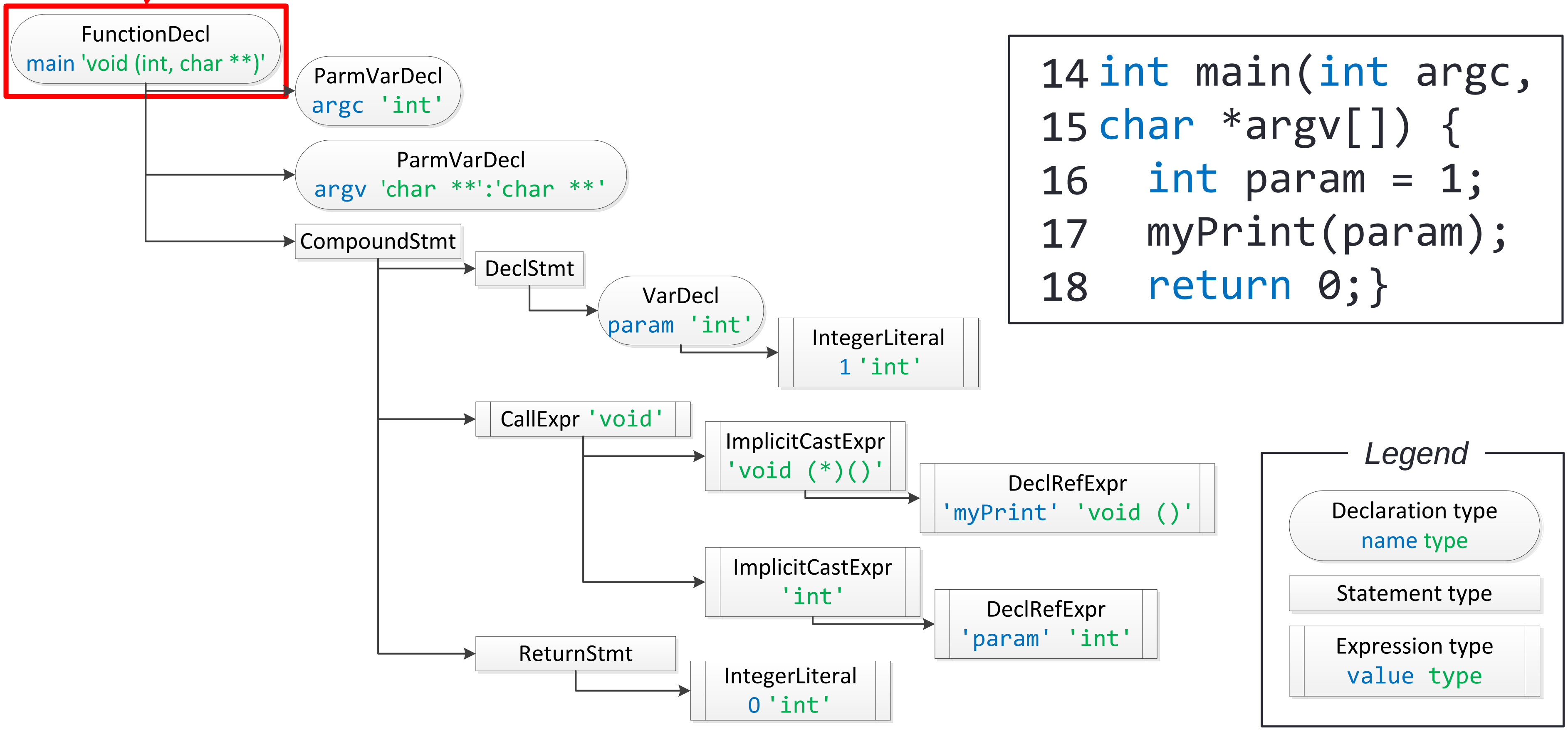
Structure of AST

- 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



FunctionDecl
main 'void (int, char **)'

ParmVarDecl
argc 'int'

ParmVarDecl
argv 'char **':'char **'

CompoundStmt

DeclStmt

VarDecl
param 'int'

IntegerLiteral
1 'int'

CallExpr 'void'

ImplicitCastExpr
'void (*)()'

DeclRefExpr
'myPrint' 'void ()'

ImplicitCastExpr
'int'

DeclRefExpr
'param' 'int'

ReturnStmt

IntegerLiteral
0 'int'

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

Legend

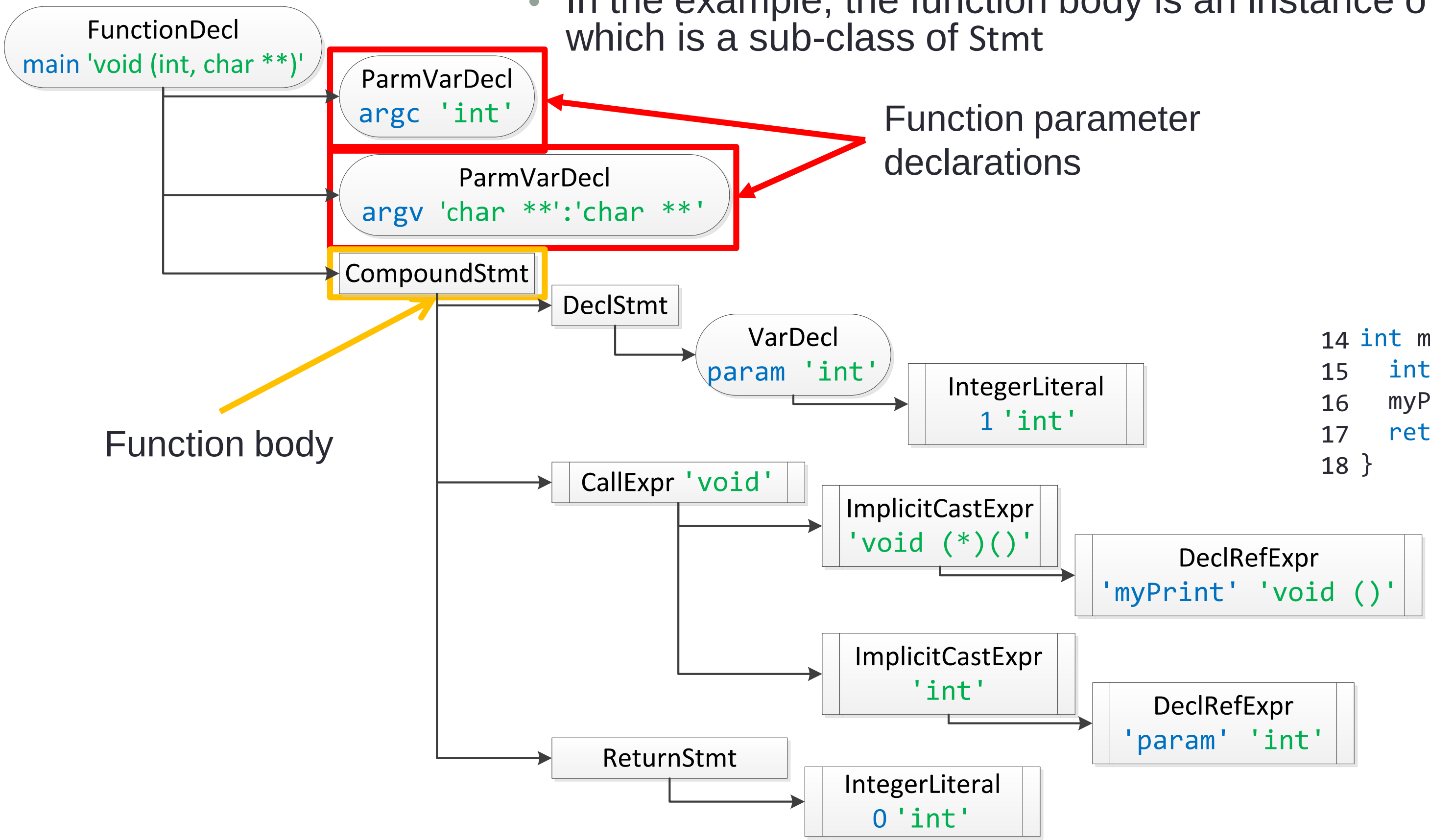
Declaration type
name type

Statement type

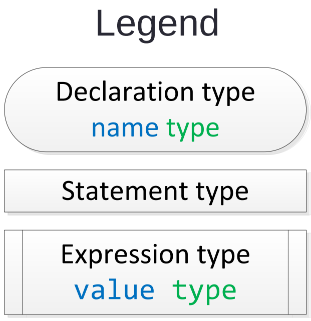
Expression type
value type

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

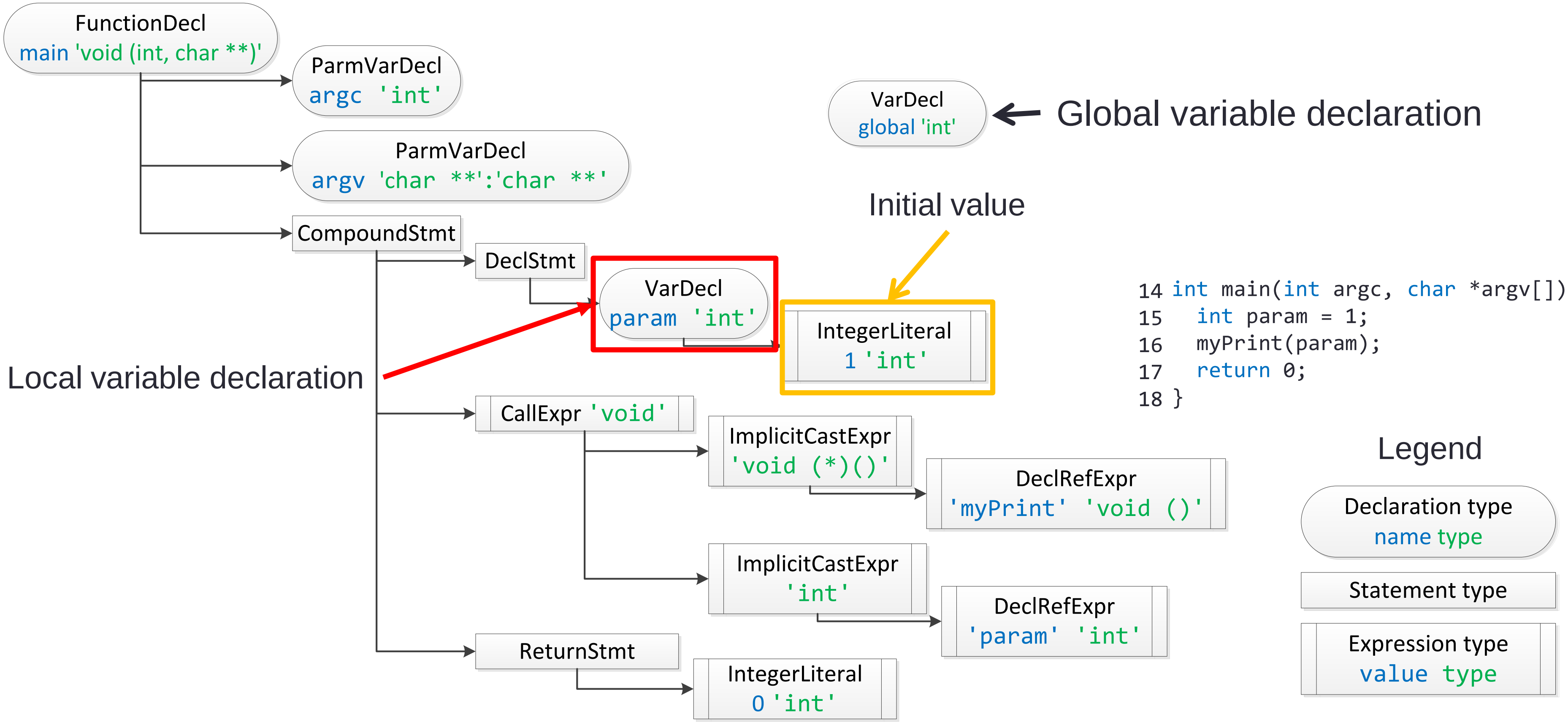


```
int main(int argc, char *argv[]) {
  int param = 1;
  myPrint(param);
  return 0;
}
```



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



FunctionDecl
main 'void (int, char **)'

ParmVarDecl
argc 'int'

ParmVarDecl
argv 'char **':'char **'

CompoundStmt

DeclStmt

VarDecl
param 'int'

IntegerLiteral
1 'int'

CallExpr 'void'

ImplicitCastExpr
'void (*)()'

DeclRefExpr
'myPrint' 'void ()'

ImplicitCastExpr
'int'

DeclRefExpr
'param' 'int'

ReturnStmt

IntegerLiteral
0 'int'

VarDecl
global 'int'

Global variable declaration

Initial value

Local variable declaration

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

Legend

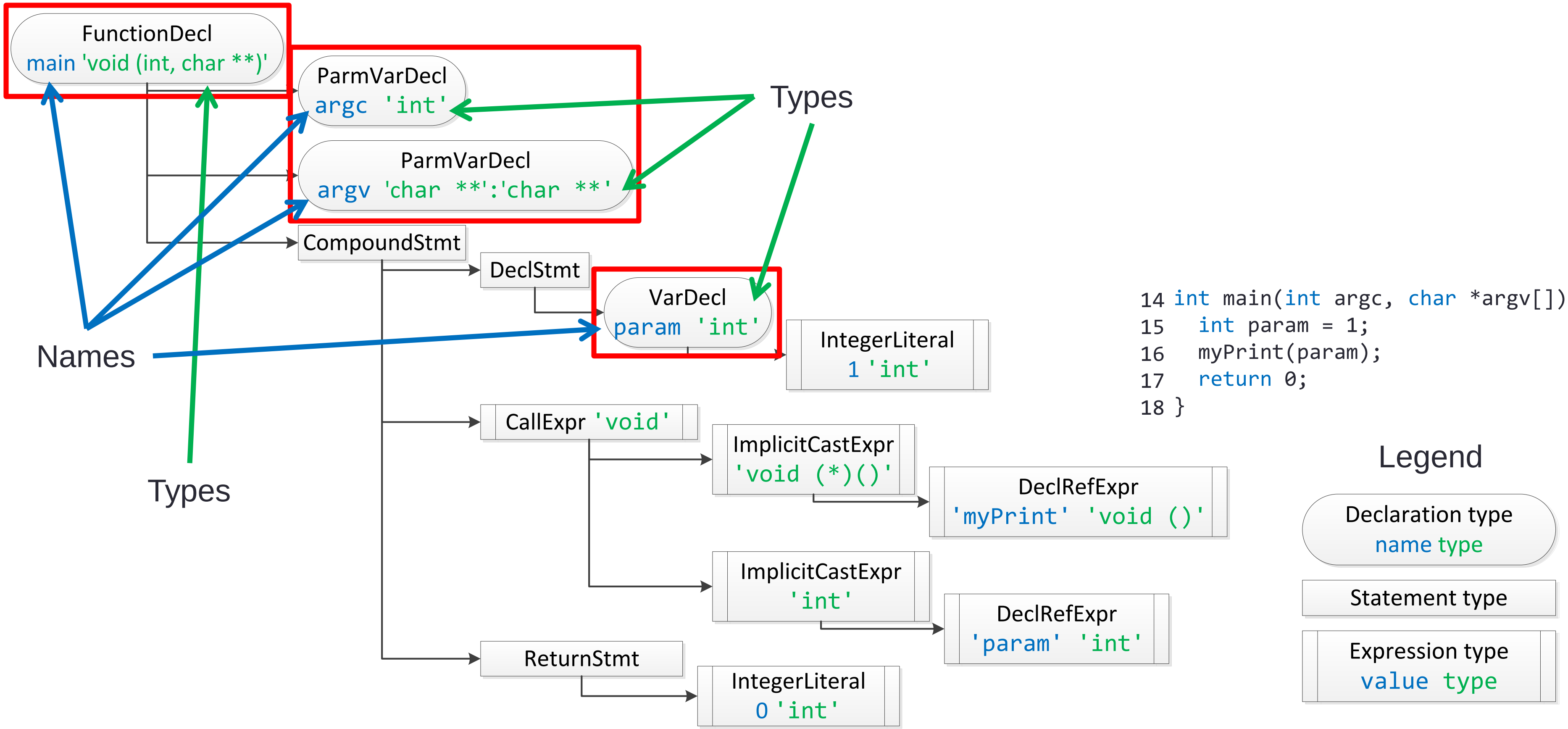
Declaration type
name type

Statement type

Expression type
value type

Decl (4/4)

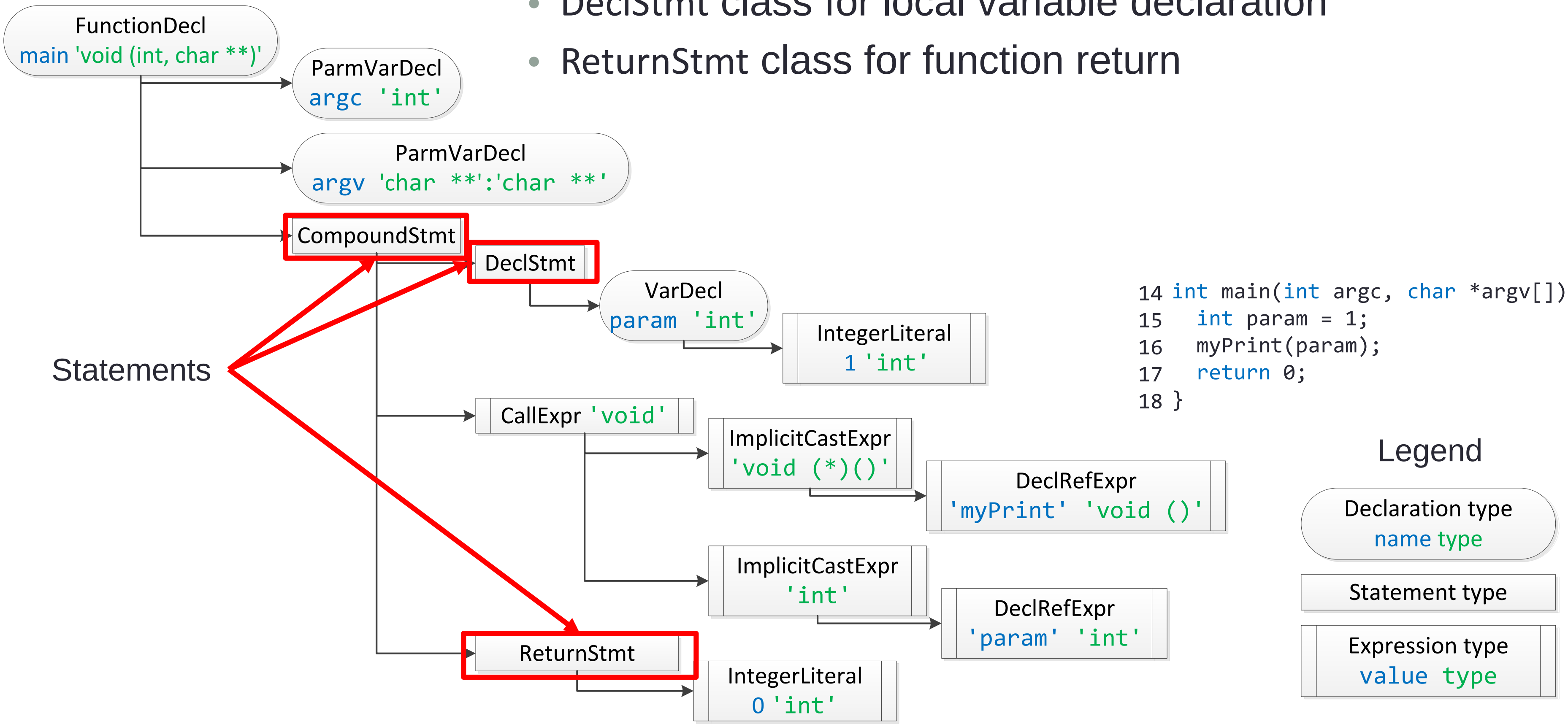
- FunctionDecl, ParmVarDecl and VarDecl have a name and a type of declaration
  - Ex) FunctionDecl has a name ‘main’ and a type ‘void (int, char**)’



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

Stmt (1/9)

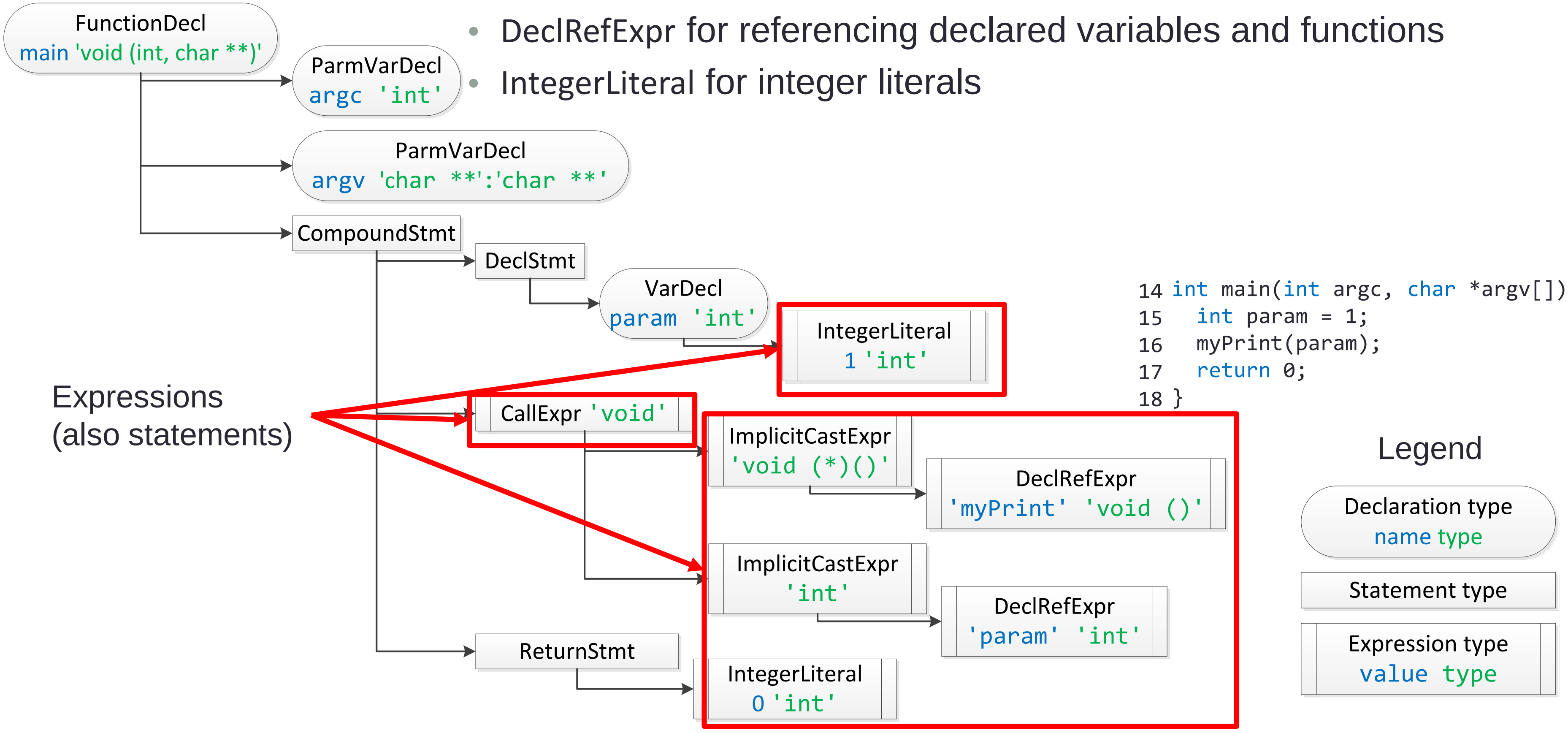
- Stmt represents a statement
  - Subclasses of Stmt
    - CompoundStmt class for code block
    - DeclStmt class for local variable declaration
    - ReturnStmt class for function return



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

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



FunctionDecl
main 'void (int, char **)'

ParmVarDecl
argc 'int'

ParmVarDecl
argv 'char **':'char **'

CompoundStmt

DeclStmt

VarDecl
param 'int'

IntegerLiteral
1 'int'

CallExpr 'void'

ImplicitCastExpr
'void (*)()'

DeclRefExpr
'myPrint' 'void ()'

ImplicitCastExpr
'int'

DeclRefExpr
'param' 'int'

ReturnStmt

IntegerLiteral
0 'int'

Expressions
(also statements)

Legend

Declaration type
name type

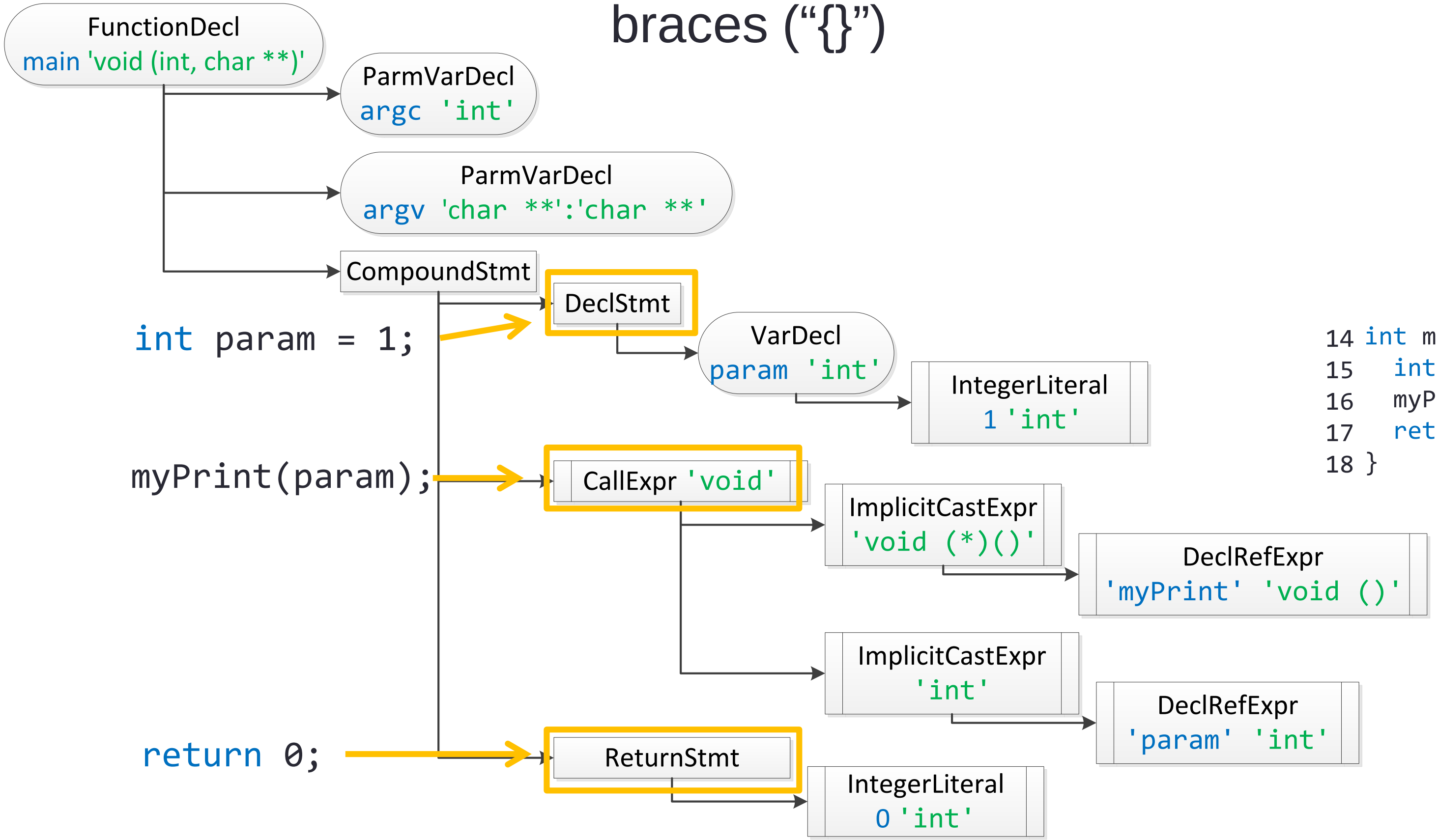
Statement type

Expression type
value type

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

Stmt (3/9)

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



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

Legend

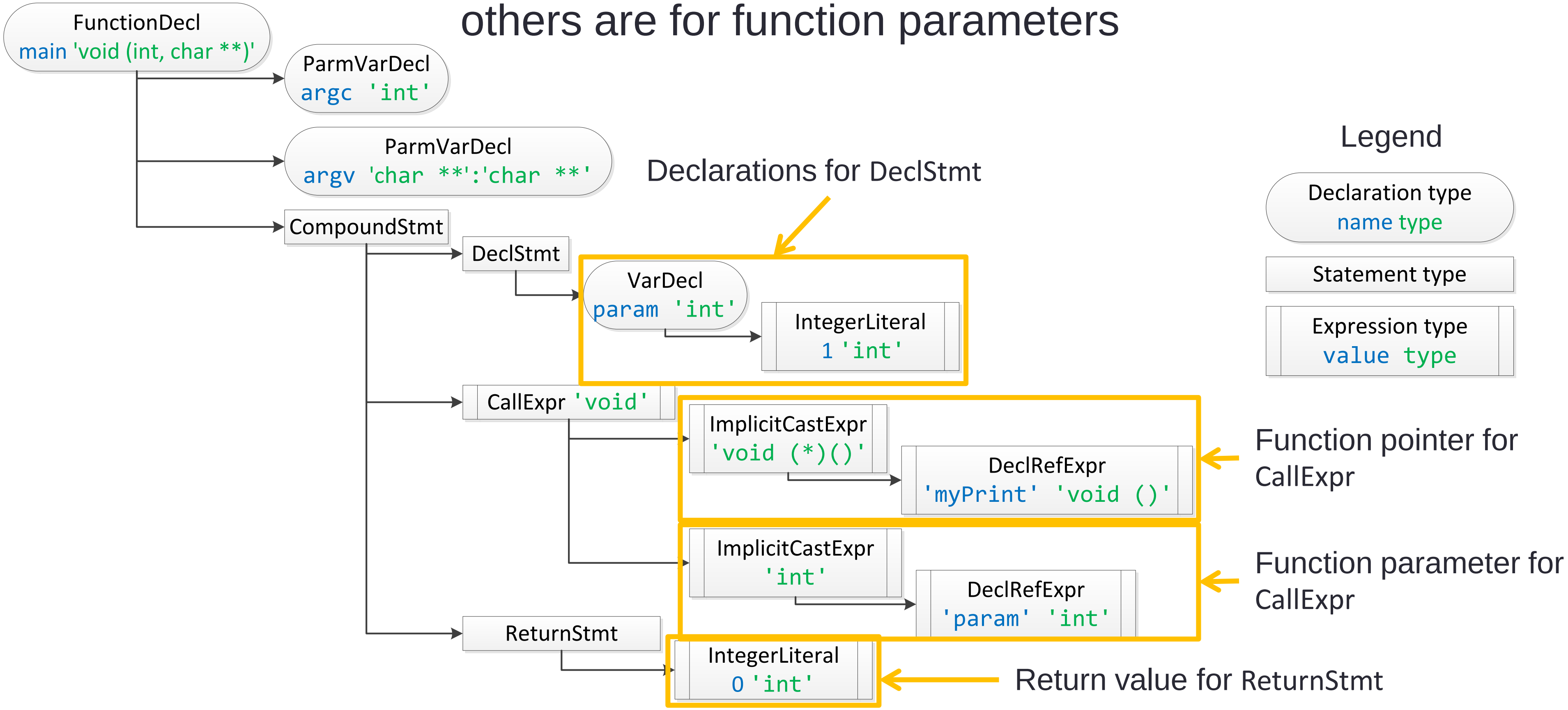
Declaration type
name type

Statement type

Expression type
value type

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



FunctionDecl
main 'void (int, char **)'

ParmVarDecl
argc 'int'

ParmVarDecl
argv 'char **':'char **'

CompoundStmt

DeclStmt

VarDecl
param 'int'

IntegerLiteral
1 'int'

CallExpr 'void'

ImplicitCastExpr
'void (*)()'

DeclRefExpr
'myPrint' 'void ()'

ImplicitCastExpr
'int'

DeclRefExpr
'param' 'int'

ReturnStmt

IntegerLiteral
0 'int'

Declarations for DeclStmt

Function pointer for CallExpr

Function parameter for CallExpr

Return value for ReturnStmt

Legend

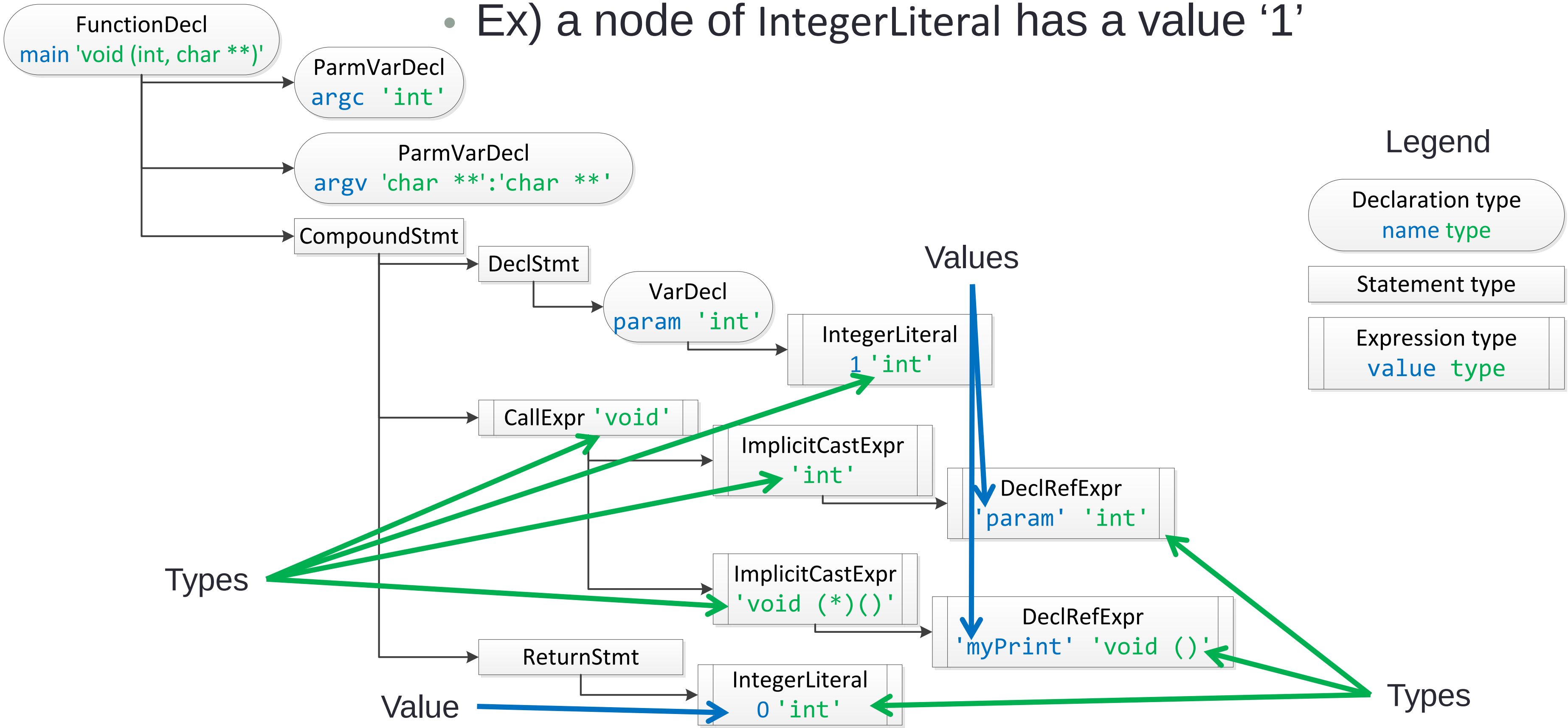
Declaration type
name type

Statement type

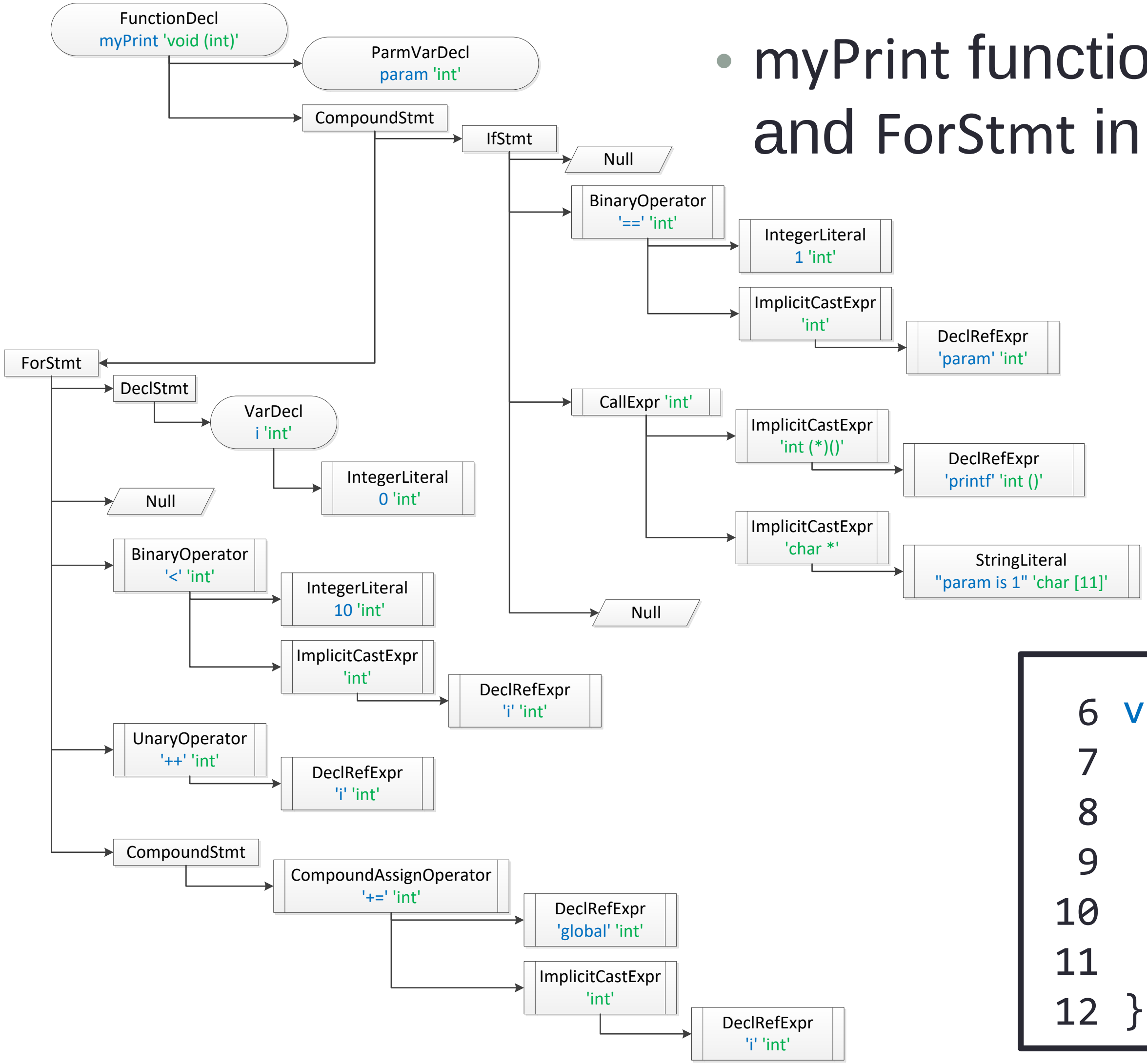
Expression type
value type

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'



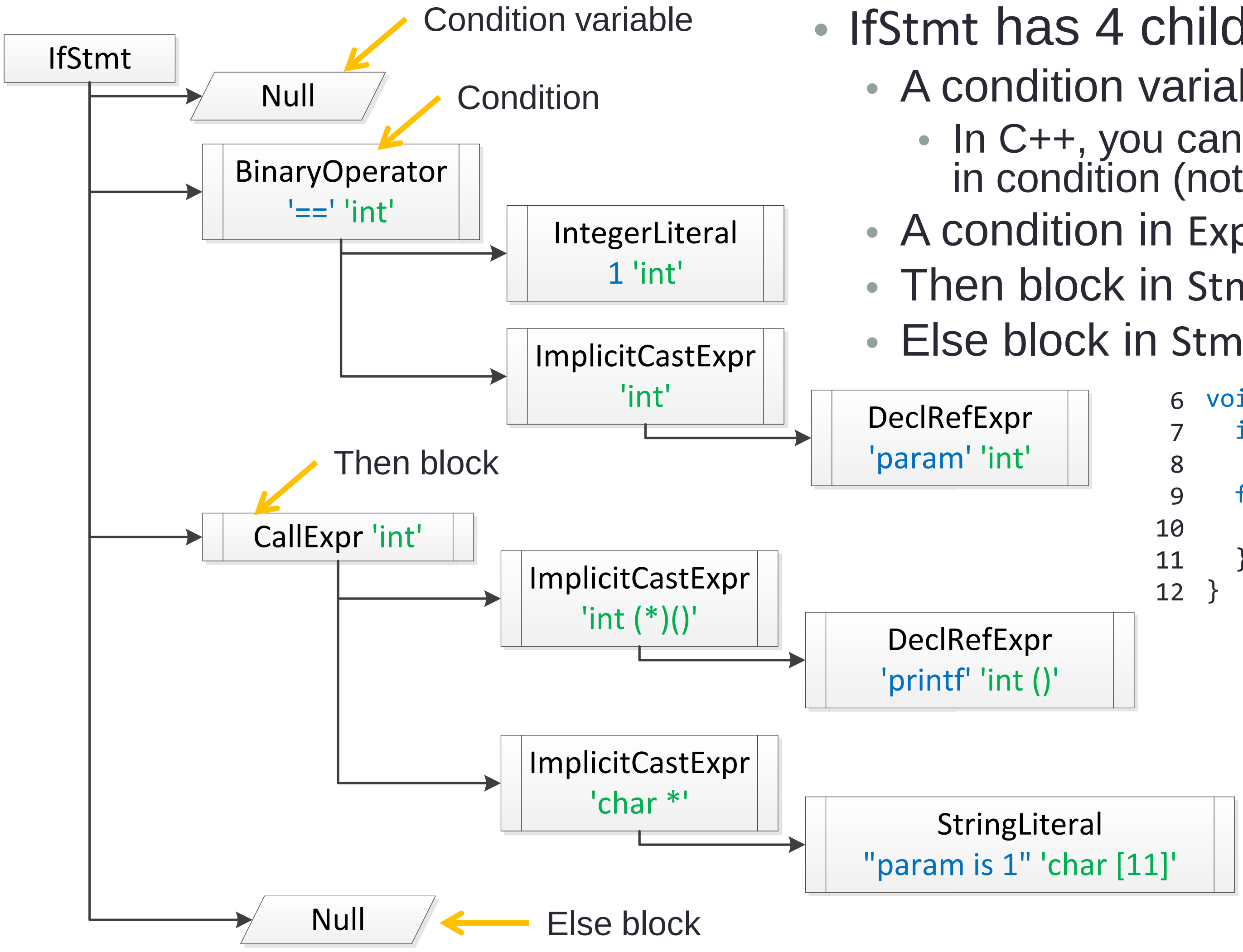
Stmt (6/9)



- myPrint function contains IfStmt and ForStmt in its function body

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

Stmt (7/9)



IfStmt

Null

Condition variable

BinaryOperator '==' 'int'

Condition

IntegerLiteral 1 'int'

ImplicitCastExpr 'int'

DeclRefExpr 'param' 'int'

CallExpr 'int'

Then block

ImplicitCastExpr 'int (*)()'

DeclRefExpr 'printf' 'int ()'

ImplicitCastExpr 'char *'

StringLiteral "param is 1" 'char [11]'

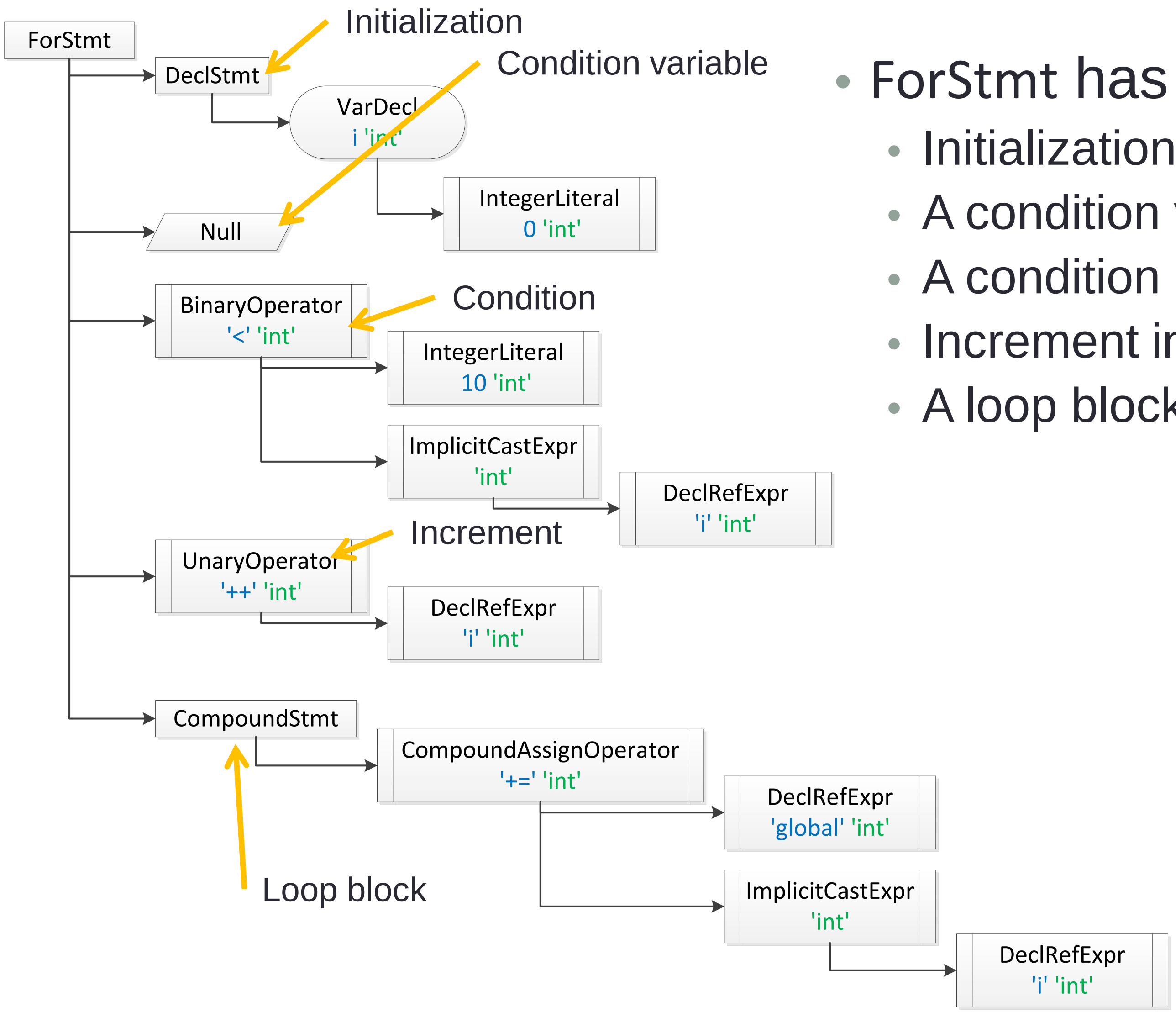
Null

Else block

- 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

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

Stmt (8/9)



ForStmt

DeclStmt

VarDecl
i 'int'

IntegerLiteral
0 'int'

Null

BinaryOperator
'<' 'int'

IntegerLiteral
10 'int'

ImplicitCastExpr
'int'

DeclRefExpr
'i' 'int'

UnaryOperator
'++' 'int'

DeclRefExpr
'i' 'int'

CompoundStmt

CompoundAssignOperator
'+=' 'int'

DeclRefExpr
'global' 'int'

ImplicitCastExpr
'int'

DeclRefExpr
'i' 'int'

Initialization

Condition variable

Condition

Increment

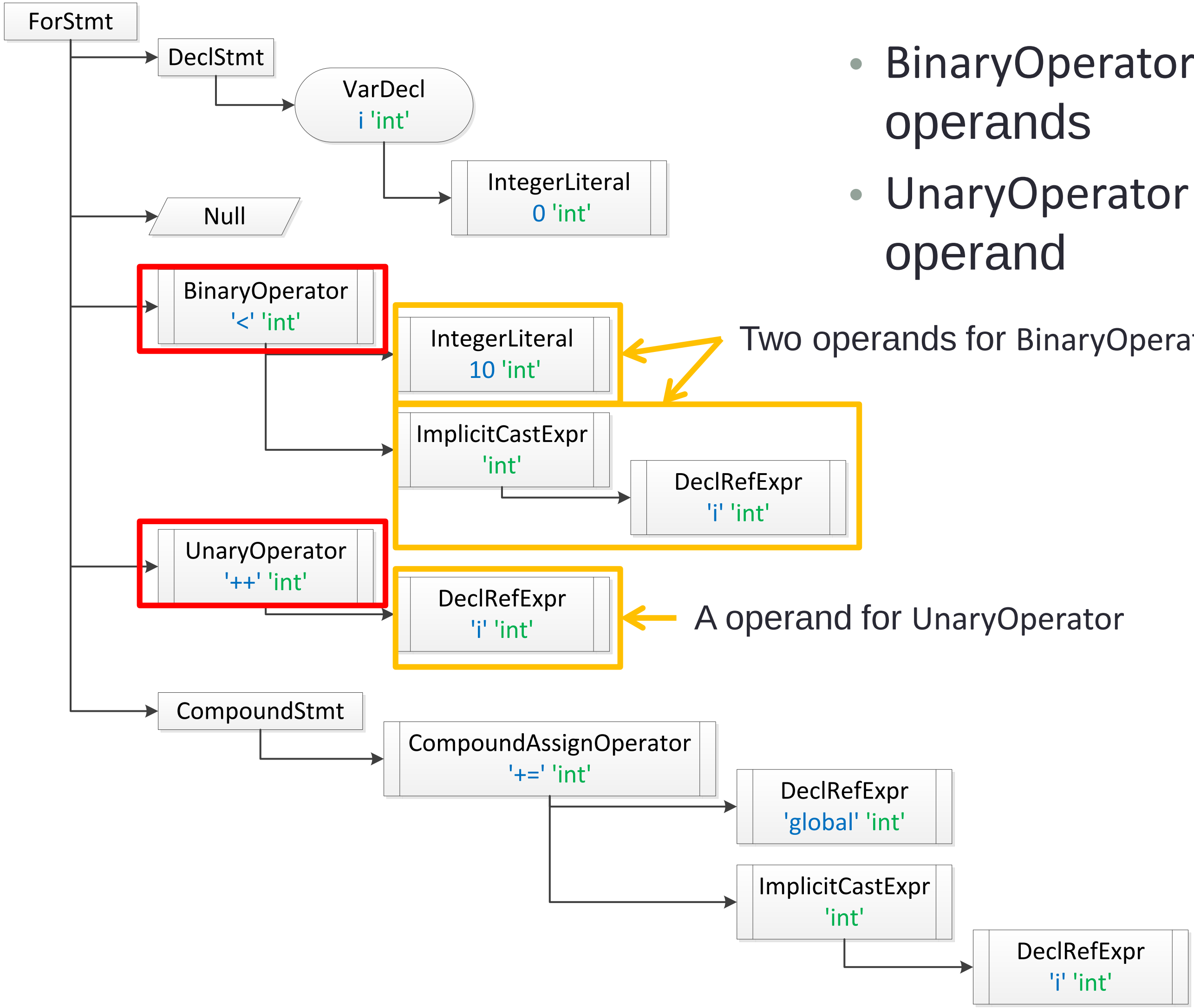
Loop block

- ForStmt has 5 children
  - Initialization in Stmt
  - A condition variable in VarDecl
  - A condition in Expr
  - Increment in Expr
  - A loop block in Stmt

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

Stmt (9/9)

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



ForStmt

DeclStmt

VarDecl
i 'int'

IntegerLiteral
0 'int'

Null

BinaryOperator
'<' 'int'

IntegerLiteral
10 'int'

ImplicitCastExpr
'int'

DeclRefExpr
'i' 'int'

UnaryOperator
'++' 'int'

DeclRefExpr
'i' 'int'

CompoundStmt

CompoundAssignOperator
'+=' 'int'

DeclRefExpr
'global' 'int'

ImplicitCastExpr
'int'

DeclRefExpr
'i' 'int'

Two operands for BinaryOperator

A operand for UnaryOperator

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

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

```
class MyASTConsumer : public ASTConsumer
{
  public:
  MyASTConsumer(Rewriter &R) {}

  virtual bool HandleTopLevelDecl(DeclGroupRef DR) {
    for(DeclGroupRef::iterator b=DR.begin(), e=DR.end(); b!=e;++b){
      … // variable b has each decleration in DR
    }
    return true;
  }
};
```

Traversing Clang AST (2/3)

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

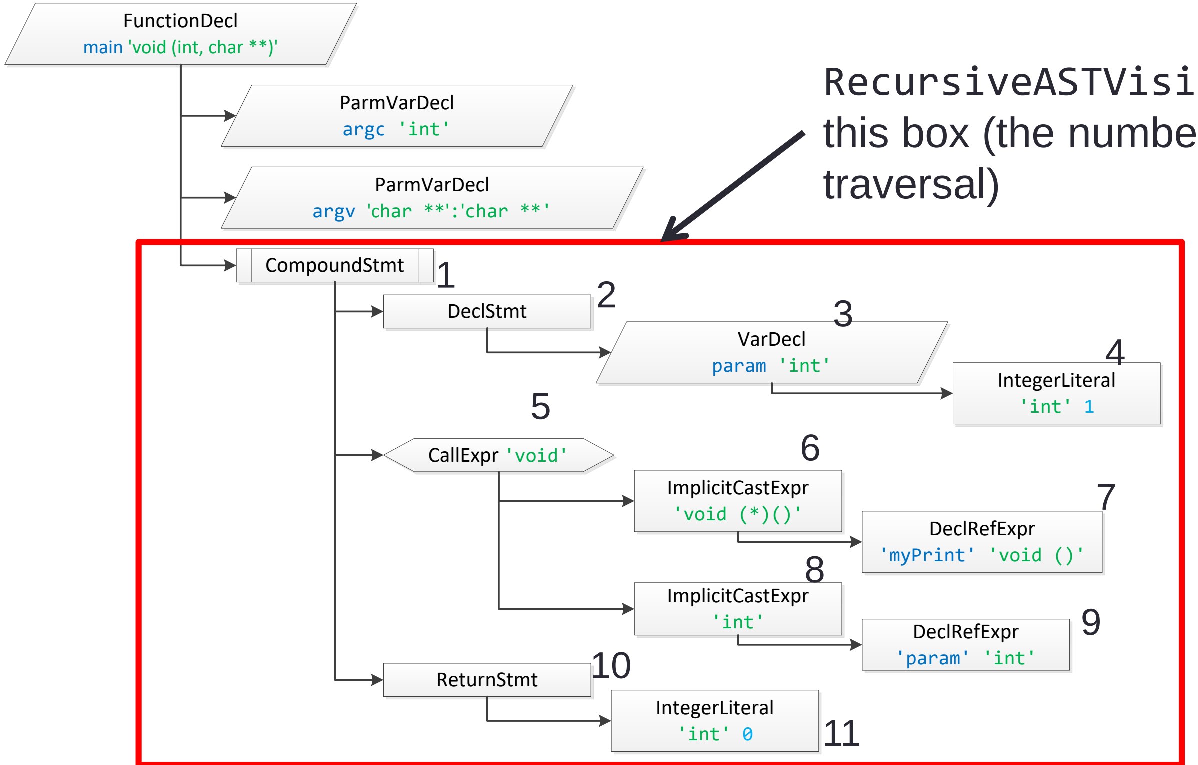
```
class MyASTVisitor : public RecursiveASTVisitor<MyASTVisitor> {
  bool VisitStmt(Stmt *s) {
    printf("\t%s \n", s->getStmtClassName() );
    return true;
  }
  bool VisitFunctionDecl(FunctionDecl *f) {
    if (f->hasBody()) {
      Stmt *FuncBody = f->getBody();
      printf("%s\n", f->getName());
    }
    return true;
  }
};
class MyASTConsumer : public ASTConsumer {
  virtual bool HandleTopLevelDecl(DeclGroupRef DR) {
    for (DeclGroupRef::iterator b = DR.begin(), e = DR.end(); b != e; ++b) {
      MyASTVisitor Visitor;
      Visitor.TraverseDecl(*b);
    }
    return true;
  }
  …
};
```

VisitStmt is called when Stmt is encountered

VisitFunctionDecl is called when FunctionDecl is encountered

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



`RecursiveASTVisitor` will visit all nodes in this box (the numbers are the order of traversal)