Clang Tutorial

CS453 Automated Software Testing

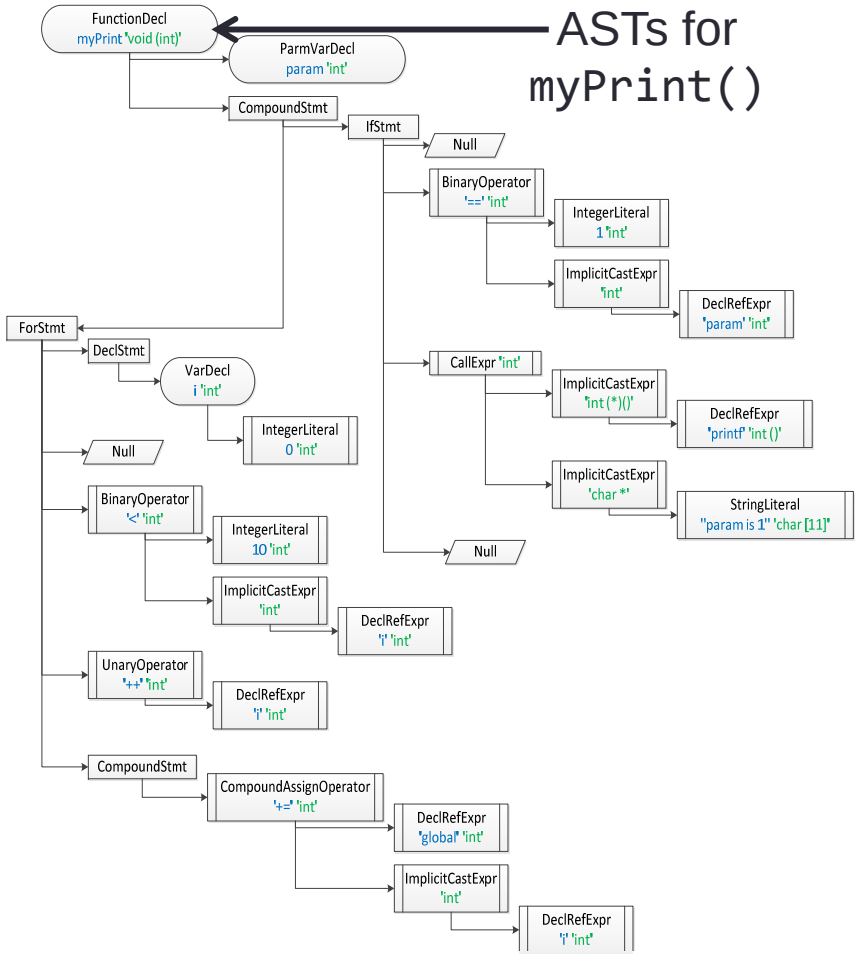
Overview

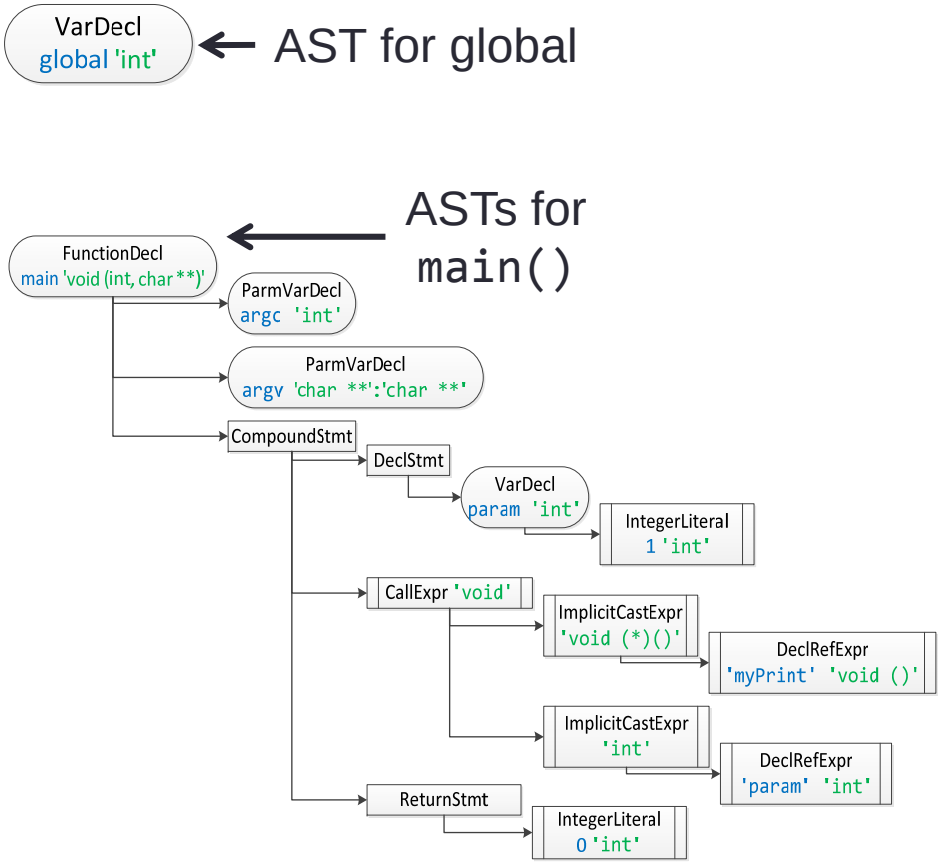
- Clang is a library to convert a C program into an abstract syntax tree (AST) and manipulate the AST
  - Ex) finding branches, renaming variables, pointer alias analysis, etc
- Example C code
  - 2 functions are declared: `myPrint` and `main`
    - `main` function calls `myPrint` and returns 0
    - `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



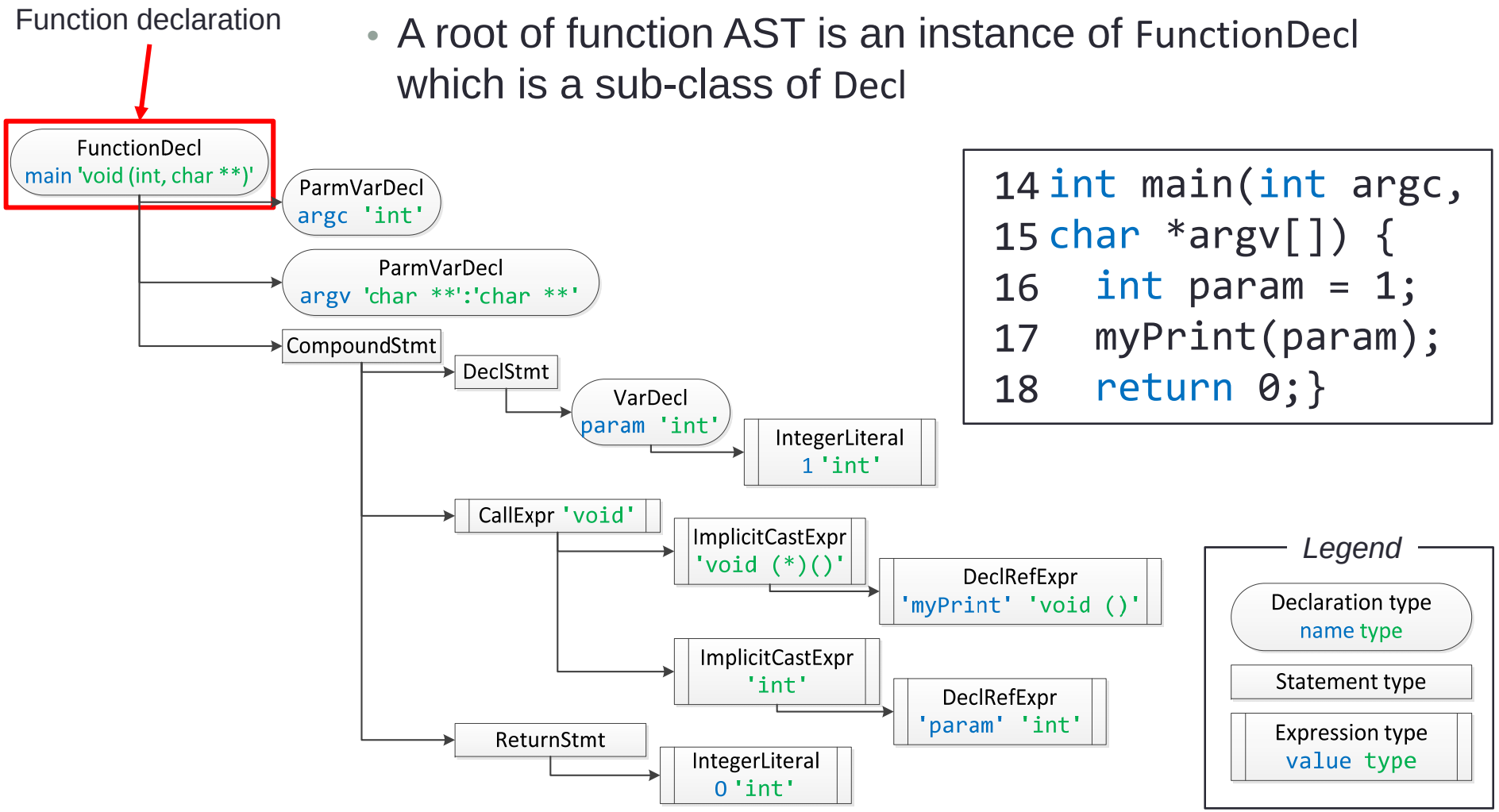


Structure of AST

- 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

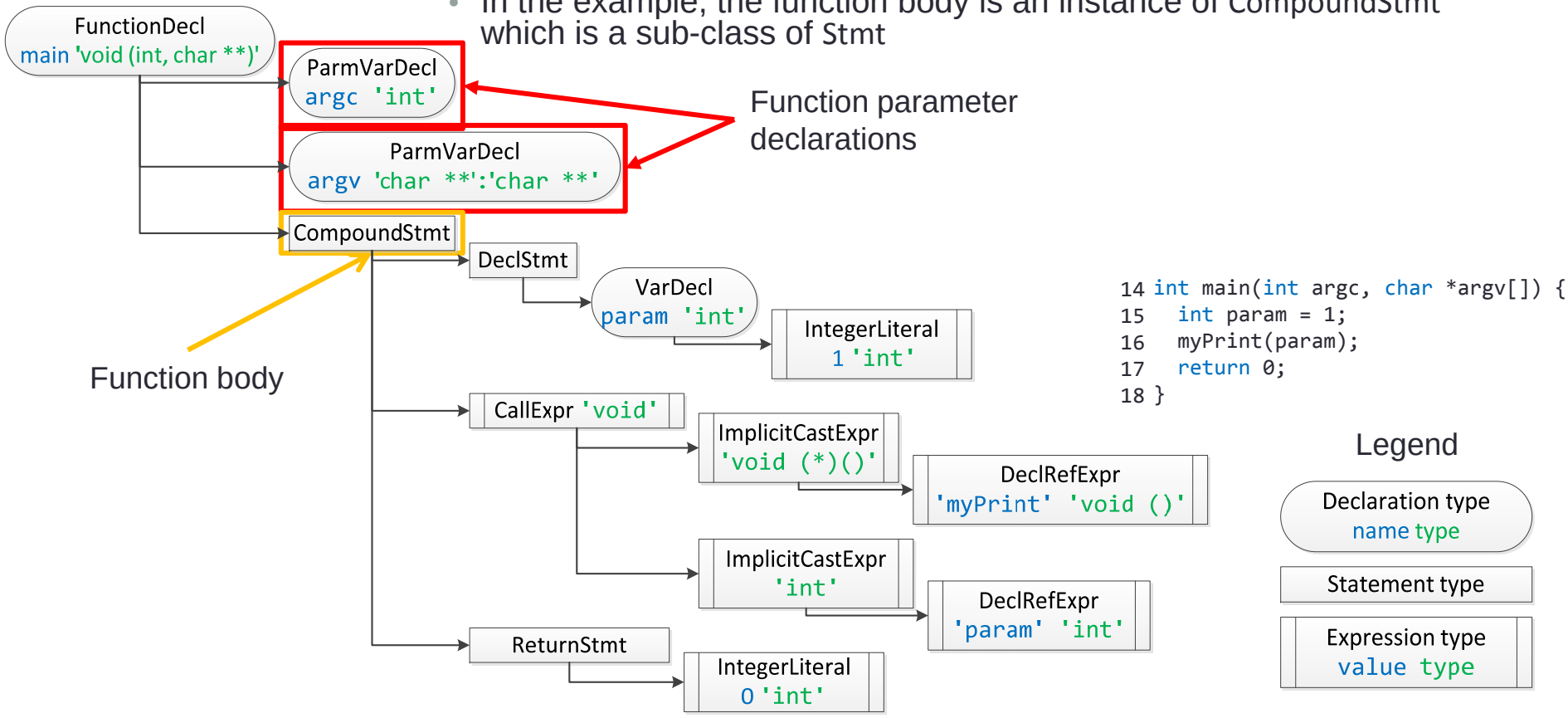
Decl (1/4)

- 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



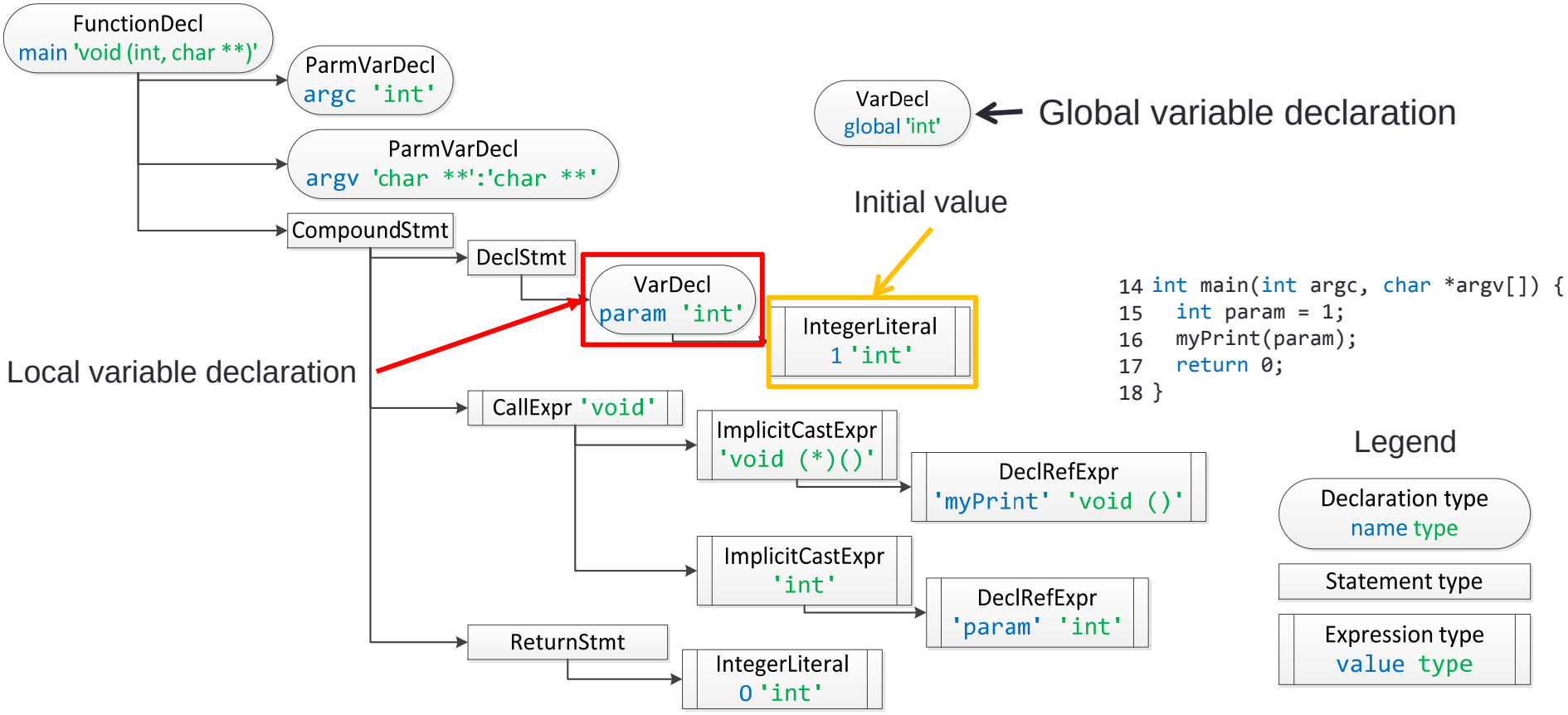
Decl (2/4)

- 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



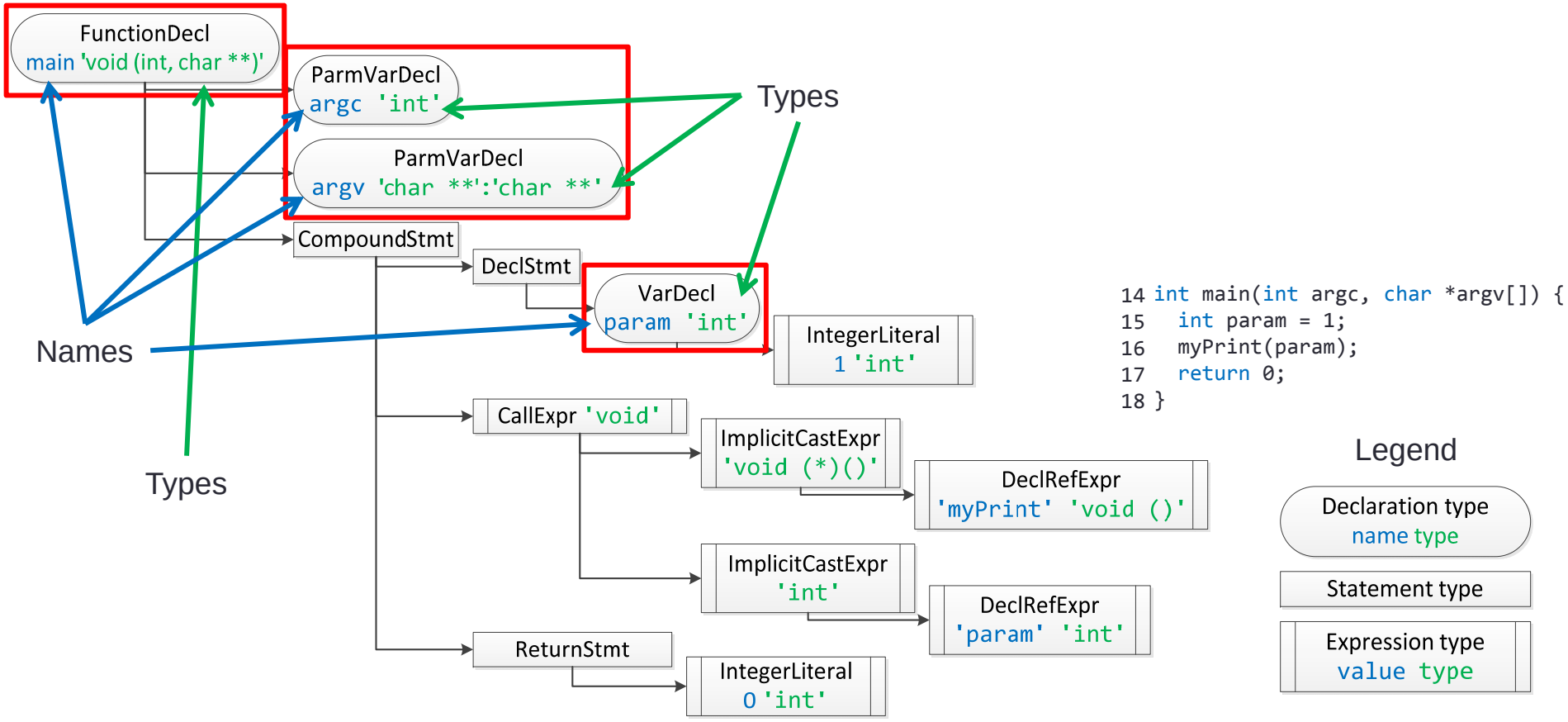
Decl (3/4)

- 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



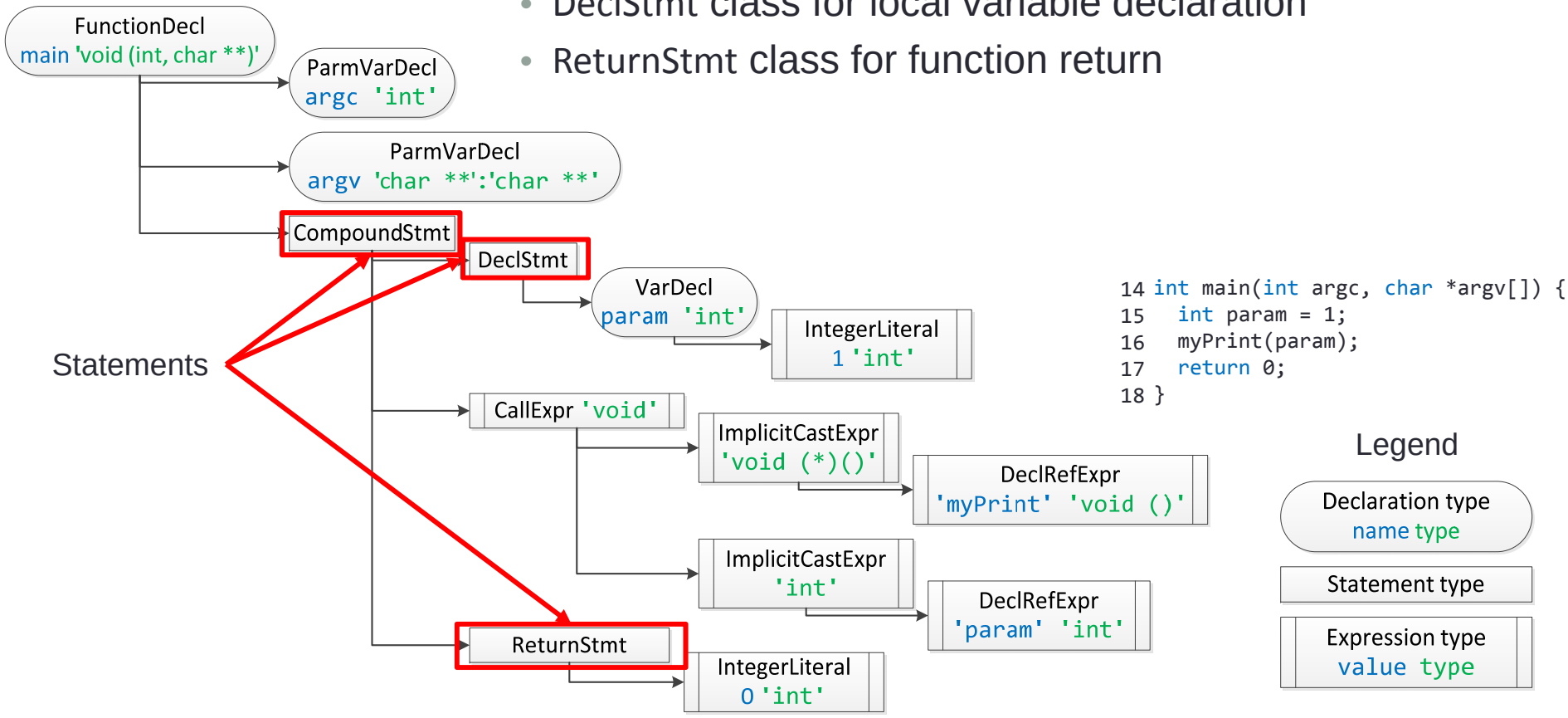
Decl (4/4)

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



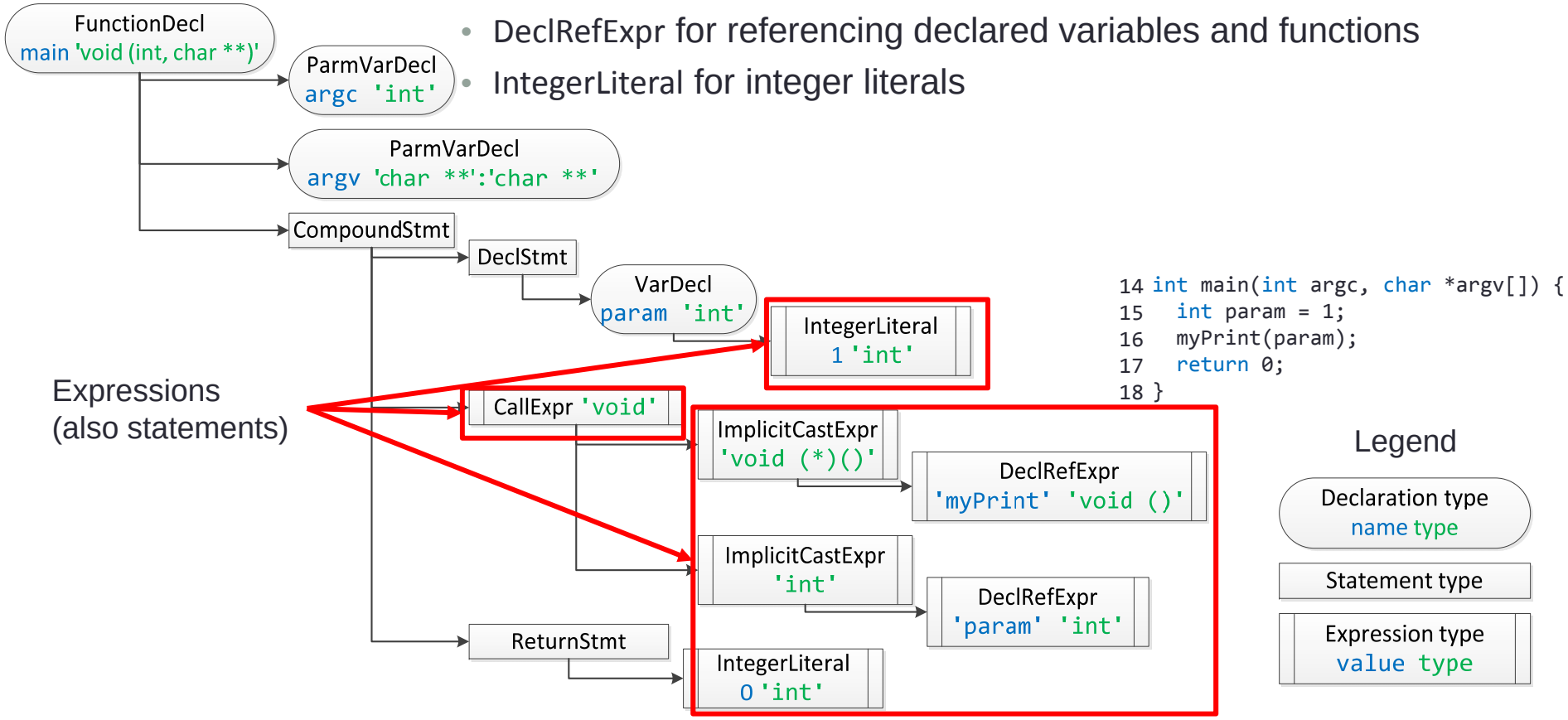
Stmt (1/9)

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



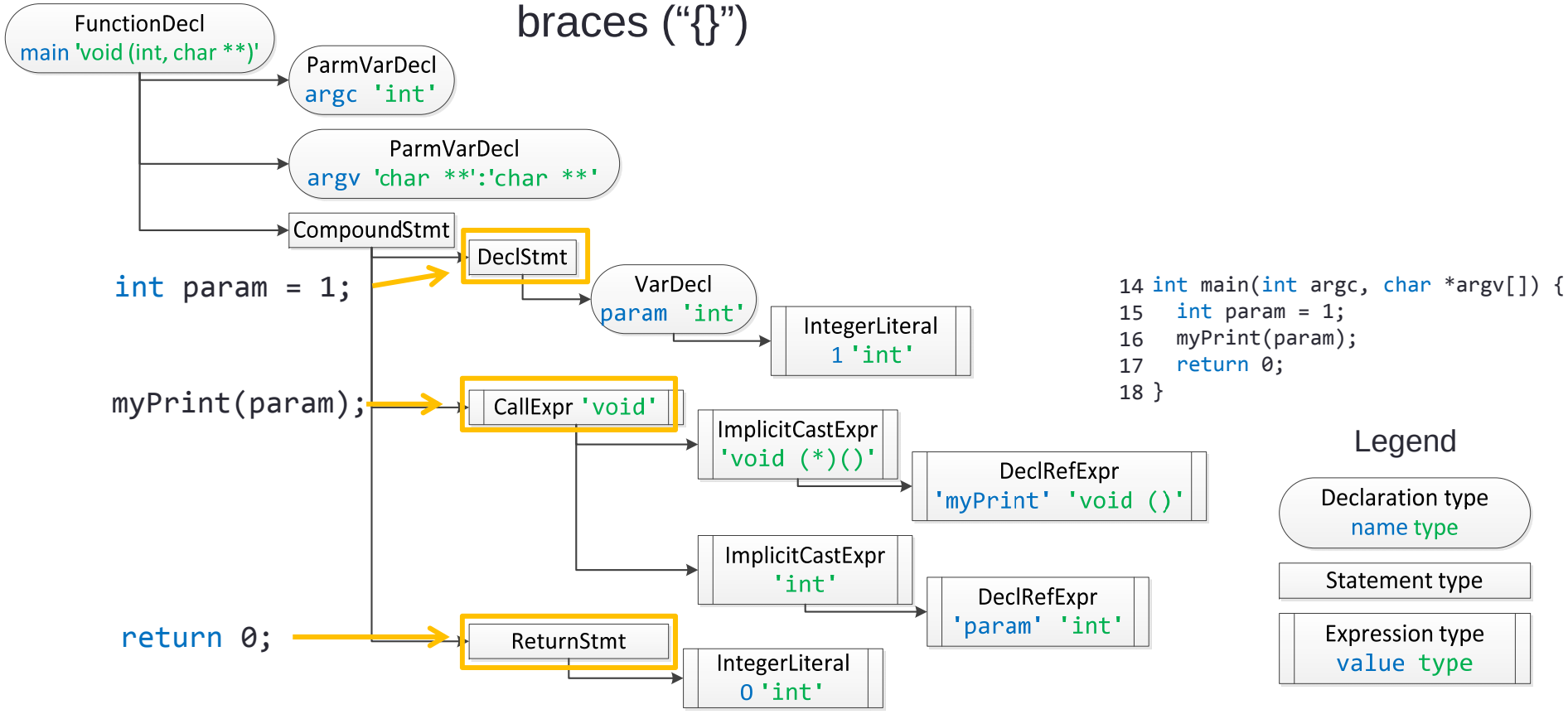
Stmt (2/9)

- 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



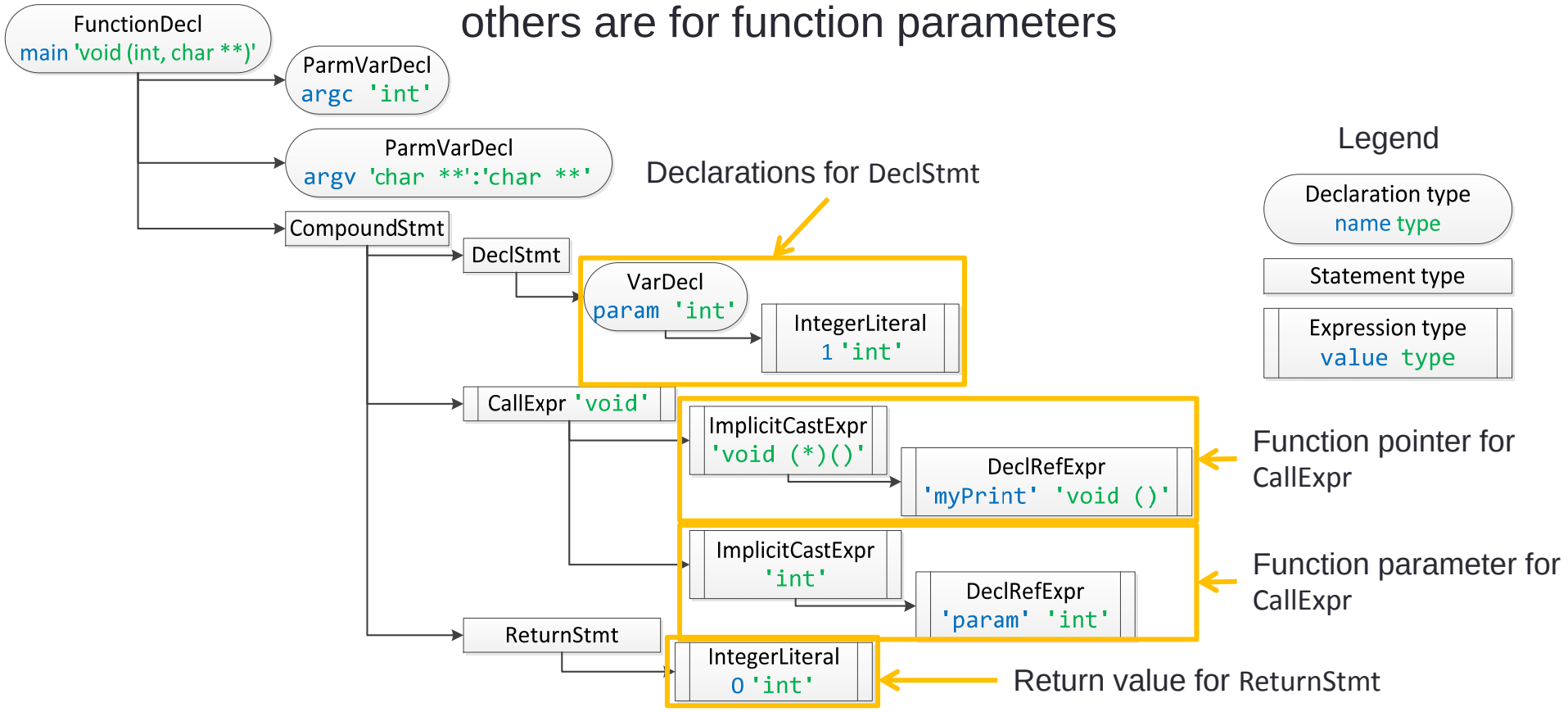
Stmt (3/9)

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



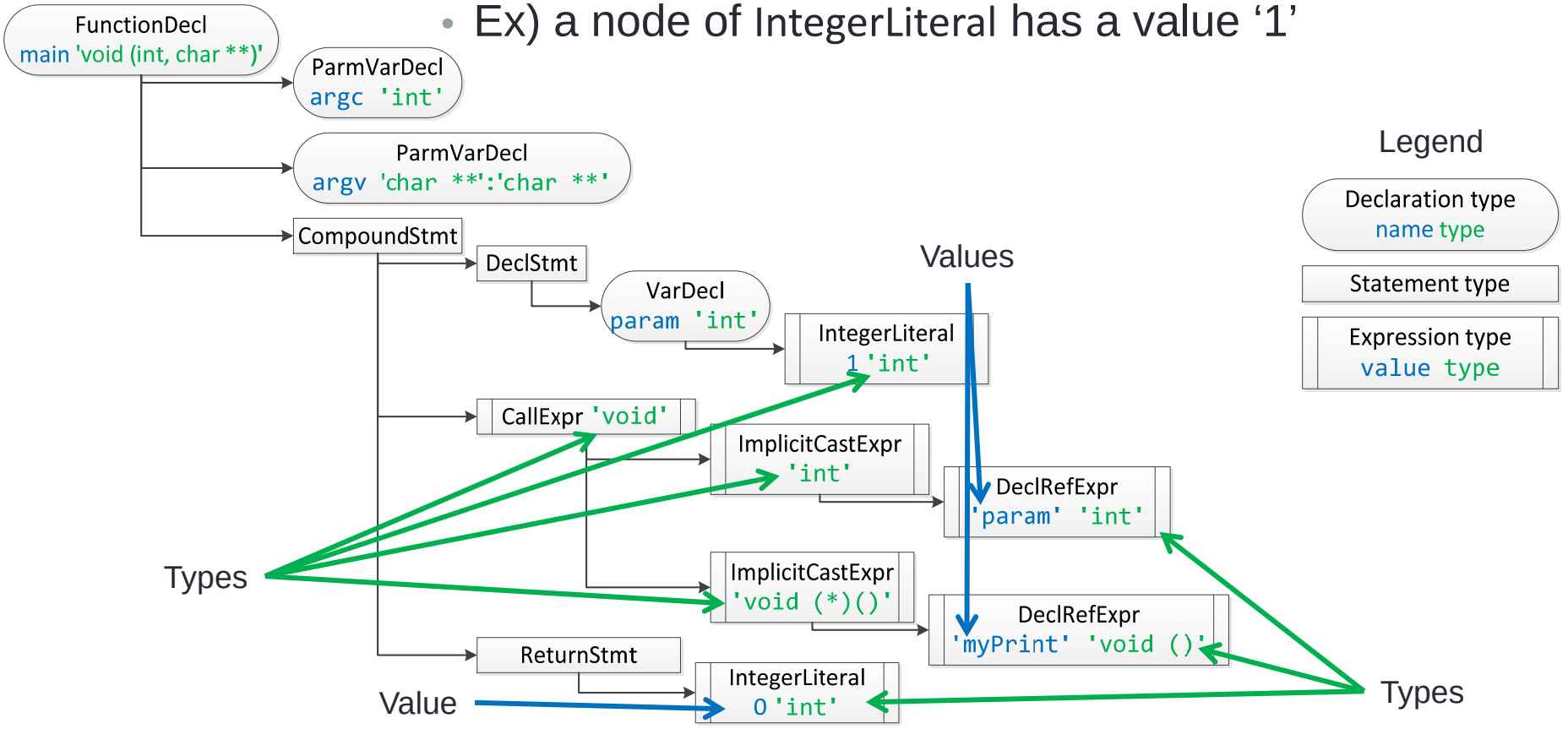
Stmt (4/9)

- 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

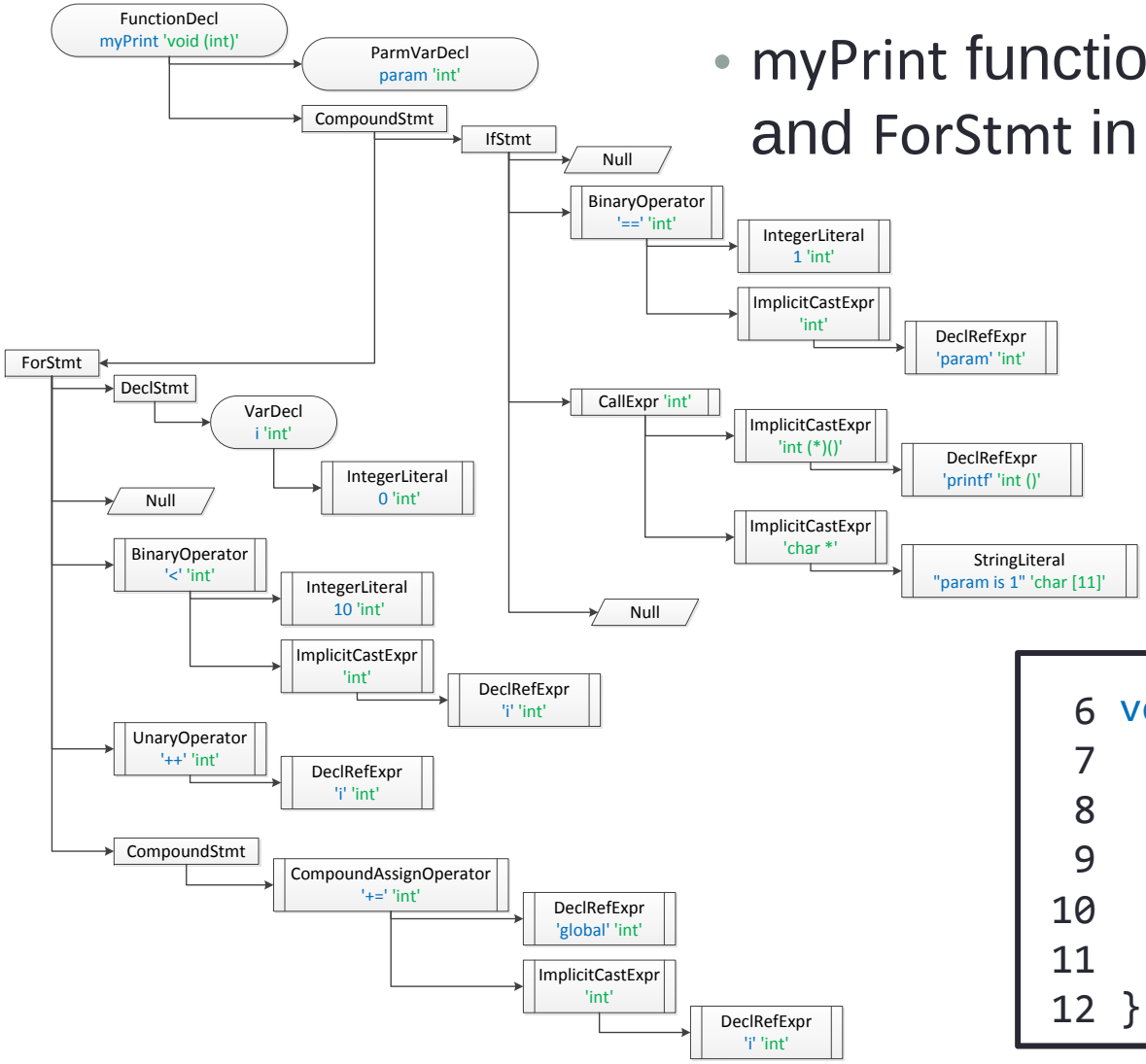


Stmt (5/9)

- 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’



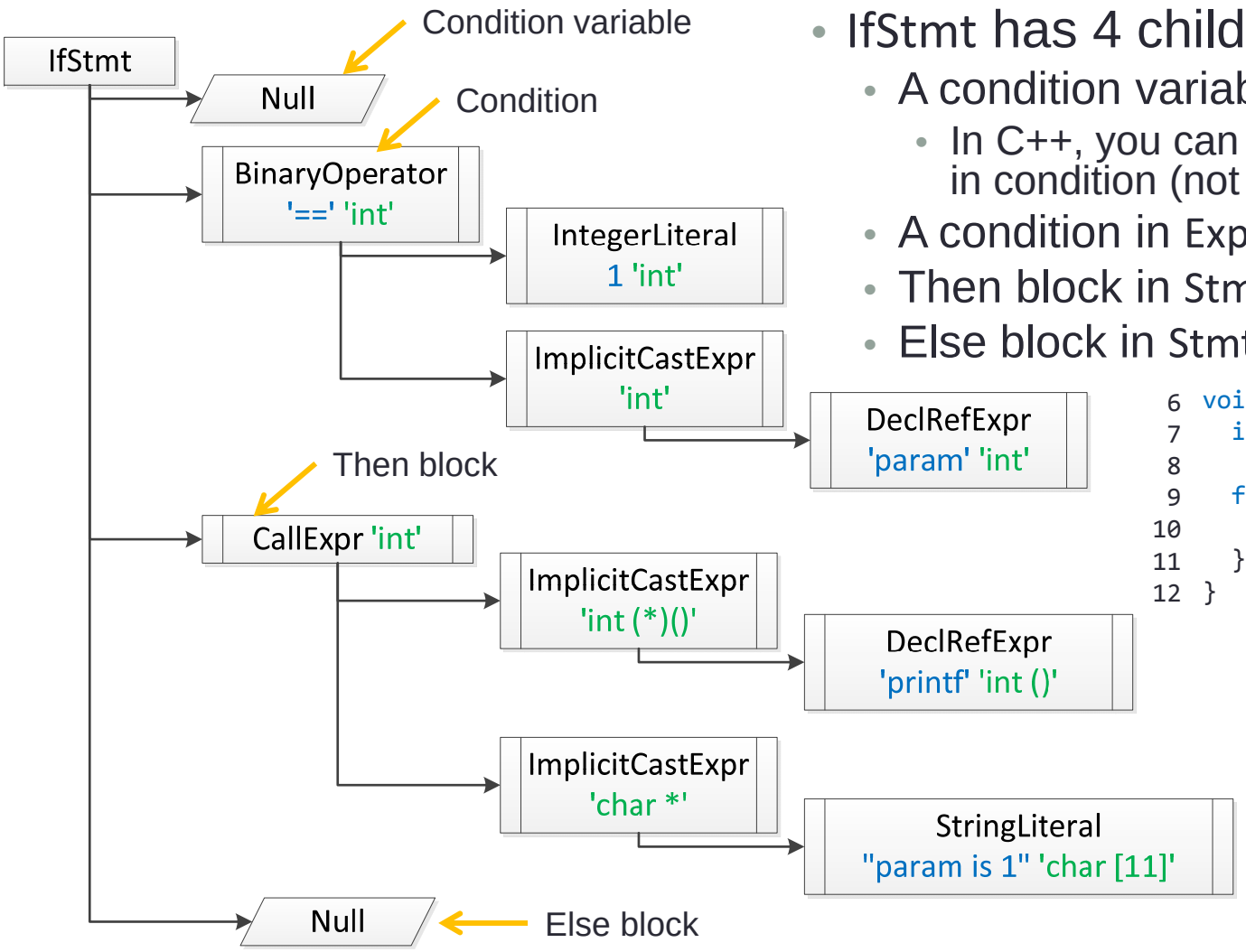
Stmt (6/9)



- 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 }
```

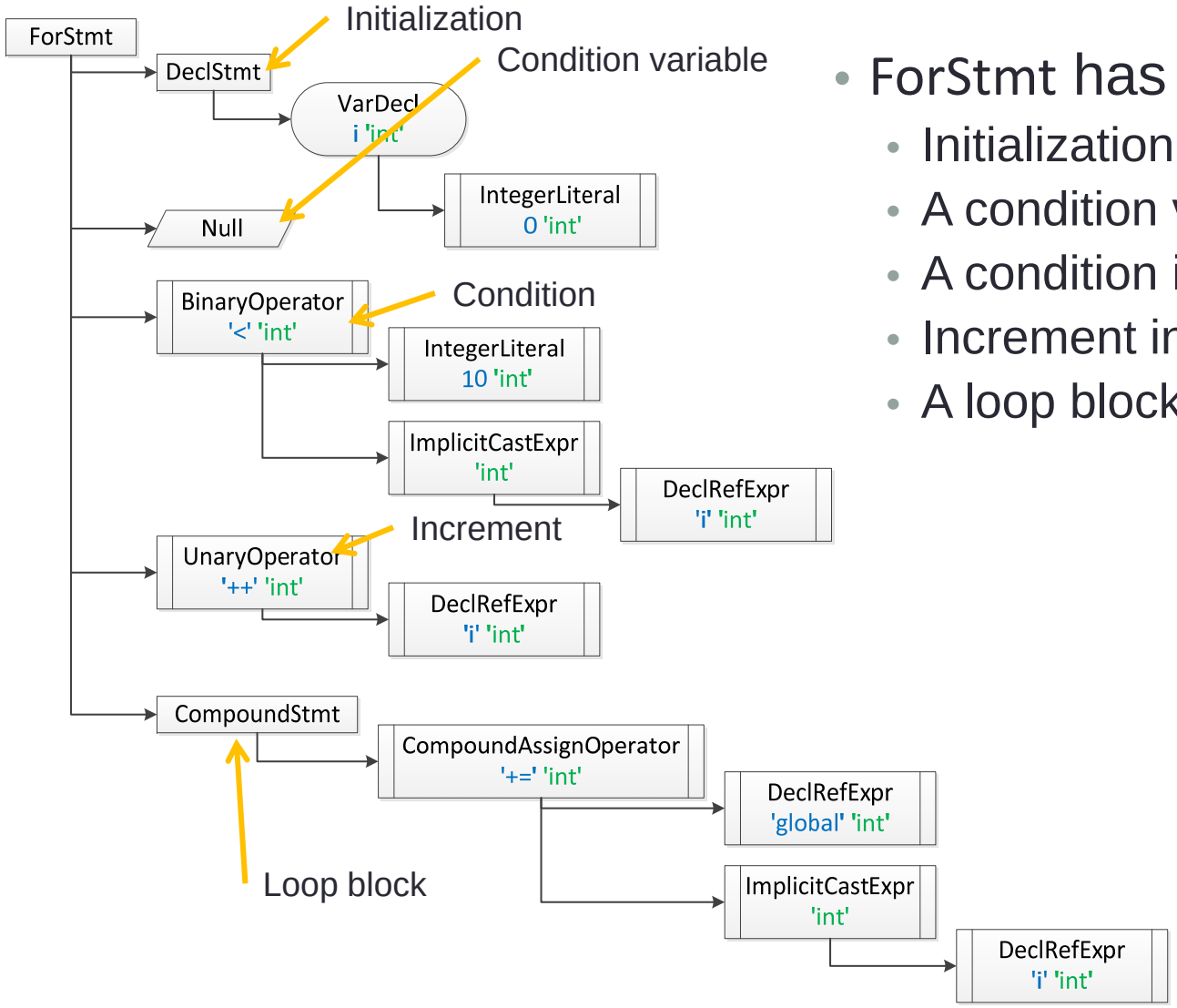
Stmt (7/9)



- 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  }
```

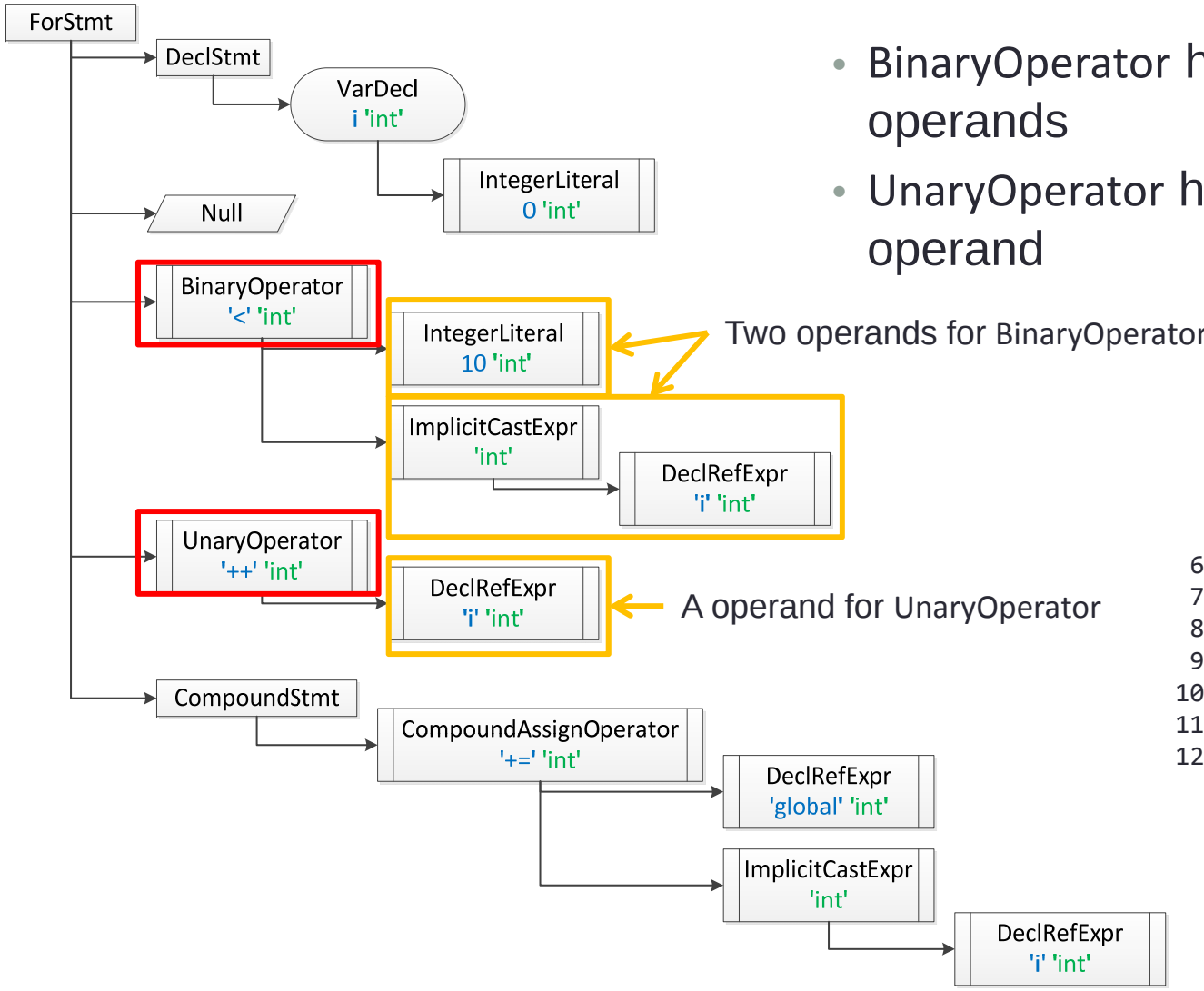
Stmt (8/9)



- 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

Two operands for BinaryOperator

A operand for UnaryOperator

```
 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)

- ParseAST() starts building and traversal of an AST
  - 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

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

- A user has to customize ASTConsumer

```
 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.

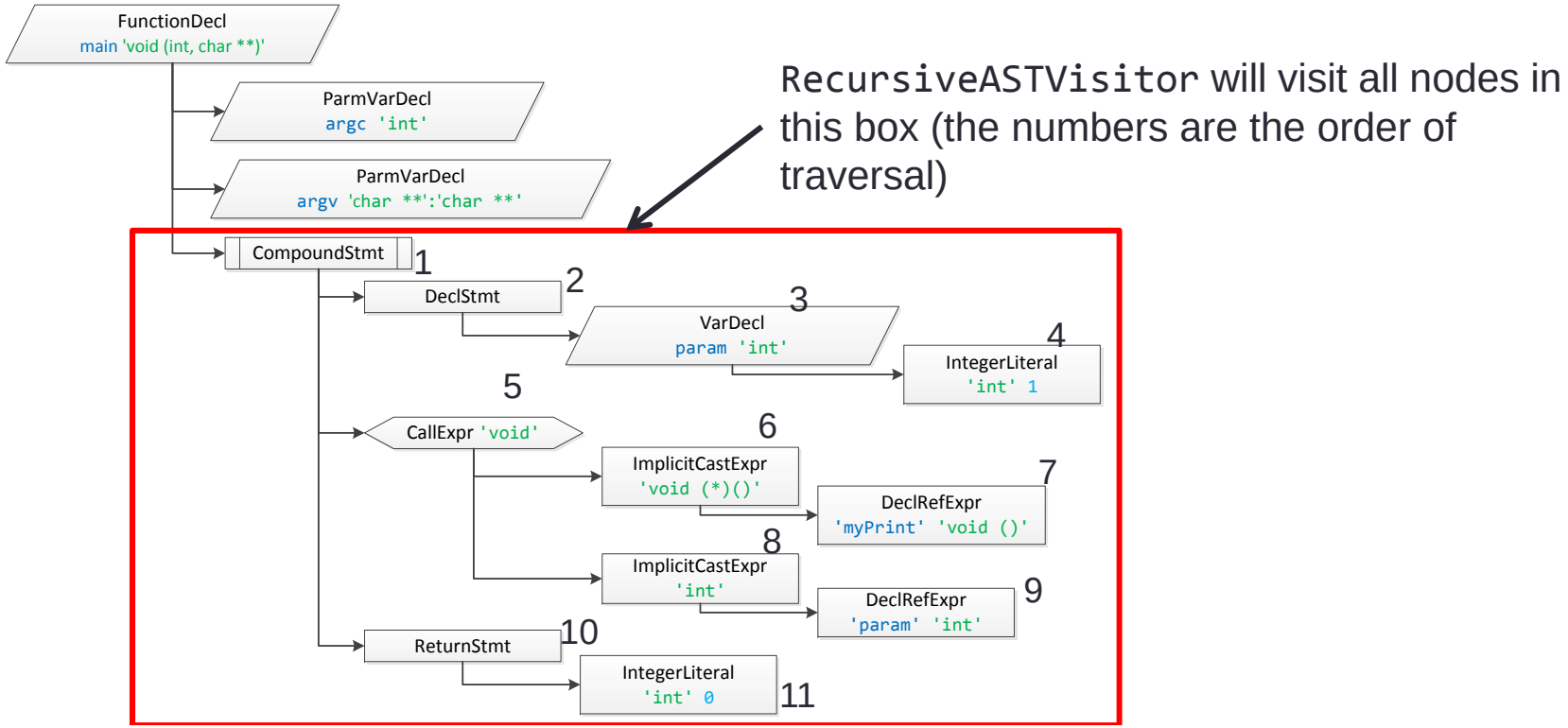
```
1  class MyASTVisitor : public RecursiveASTVisitor<MyASTVisitor> {
2    bool VisitStmt(Stmt *s) {
3      printf("\t%s \n", s->getStmtClassName() );
4      return true;
5    }
6    bool VisitFunctionDecl(FunctionDecl *f) {
7      if (f->hasBody()) {
8        Stmt *FuncBody = f->getBody();
9        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 };
```

`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



Guideline for HW #2

- Initialization of Clang
- Line number information of `Stmt`
- Useful Functions

Initialization of Clang

- Initialization of Clang is complicated
  - To use Clang, many classes should be created and many functions should be called to initialize Clang environment
    - Ex) `ComplierInstance`, `TargetOptions`, `FileManager`, etc.
- It is recommended to use the initialization part of the sample source code from the course homepage *as is*, and implement your own `ASTConsumer` and `RecursiveASTVisitor` classes

Line number information of `Stmt`

- A `SourceLocation` object from `getLocStart()` of `Stmt` has a line information
  - `SourceManager` is used to get line and column information from `SourceLocation`
    - In the initialization step, `SourceManager` object is created
    - `getExpansionLineNumber()` and `getExpansionColumnNumber()` in `SourceManager` give line and column information, respectively

```
bool VisitStmt(Stmt *s) {
  SourceLocation startLocation = s->getLocStart();
  SourceManager &srcmgr=m_srcmgr;//you can get SourceManager from the initialization part
  unsigned int lineNum = srcmgr.getExpansionLineNumber(startLocation);
  unsigned int colNum = srcmgr.getExpansionColumnNumber(startLocation);
  …
}
```

Useful Functions

- `dump()` and `dumpColor()` in `Stmt` and `FunctionDecl` to print AST
  - `dump()` shows AST rooted at `Stmt` or `FunctionDecl` object
  - `dumpColor()` is similar to `dump()` but shows AST with syntax highlight
  - Example: dumpColor() of myPrint

```
FunctionDecl 0x368a1e0 <line:6:1> myPrint 'void (int)'
|-ParmVarDecl 0x368a120 <line:3:14, col:18> param 'int'
`-CompoundStmt 0x36a1828 <col:25, line:6:1>
  `-IfStmt 0x36a17f8 <line:4:3, line:5:24>
    |-<<<NULL>>>
    |-BinaryOperator 0x368a2e8 <line:4:7, col:16> 'int' '=='
    | |-ImplicitCastExpr 0x368a2d0 <col:7> 'int' <LValueToRValue>
    | | `-DeclRefExpr 0x368a288 <col:7> 'int' lvalue ParmVar 0x368a120 'param' 'int'
    | `-IntegerLiteral 0x368a2b0 <col:16> 'int' 1
    |-CallExpr 0x368a4e0 <line:5:5, col:24> 'int'
    | |-ImplicitCastExpr 0x368a4c8 <col:5> 'int (*)()' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x368a400 <col:5> 'int ()' Function 0x368a360 'printf' 'int ()'
    | `-ImplicitCastExpr 0x36a17e0 <col:12> 'char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x368a468 <col:12> 'char [11]' lvalue "param is 1"
    `-<<<NULL>>>
```

Guideline for HW #3

- Code modification using `Rewriter`
- Converting `Stmt` into String
- Obtaining `SourceLocation`

Code Modification using Rewriter

- You can modify code using Rewriter class
  - Rewriter has functions to insert, remove and replace code
    - InsertTextAfter(*loc,str*), InsertTextBefore(*loc,str*), RemoveText(*loc,size*), ReplaceText(…) , etc. where loc, str, size are a location (SourceLocation), a string, and a size of statement to remove, respectively
- Example: inserting a text before a condition in IfStmt using InsertTextAfter()

```
1 bool MyASTVisitor::VisitStmt(Stmt *s) {
2   if (isa<IfStmt>(s)) {
3     IfStmt *ifStmt = cast<IfStmt>(s);
4     condition = ifStmt->getCond();
5     m_rewriter.InsertTextAfter(condition->getLocStart(), "/*start of cond*/");
6   }
7 }
```

```
if( param == 1 ) ⟶ if( /*start of cond*/param == 1 )
```

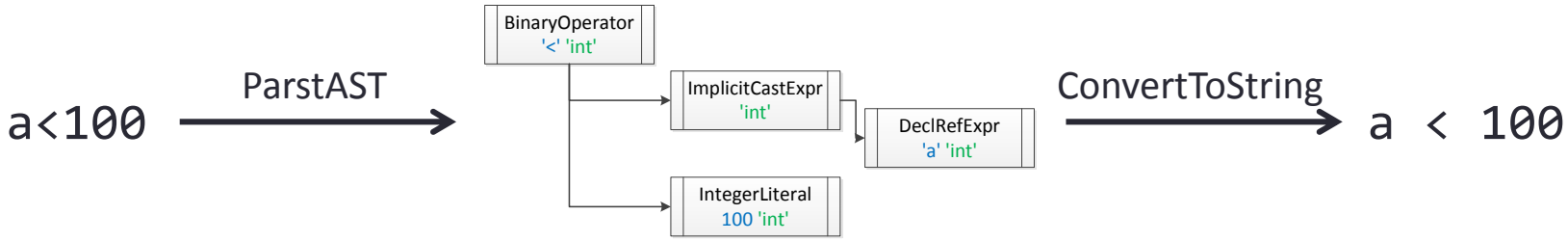
Output of Rewriter

- Modified code is obtained from a RewriterBuffer of Rewriter through getRewriteBufferFor()
- Example code which writes modified code in output.txt
  - ParseAST() modifies a target code as explained in the previous slides
    - TheConsumer contains a Rewriter instance TheRewriter

```
1 int main(int argc, char *argv[]) {
2   …
3   ParseAST(TheCompInst.getPreprocessor(), &TheConsumer, TheCompInst.getASTContext());
4   const RewriteBuffer *RewriteBuf = TheRewriter.getRewriteBufferFor(SourceMgr.getMainFileID());
5   ofstream output(“output.txt”);
6   output << string(RewriteBuf->begin(), RewriteBuf->end());
7   output.close();
8 }
```

Converting Stmt into String

- ConvertToString(stmt) of Rewriter returns a string corresponding to Stmt
  - The returned string may not be exactly same to the original statement since ConvertToString() prints a string using the Clang pretty printer
    - For example, ConvertToString() will insert a space between an operand and an operator



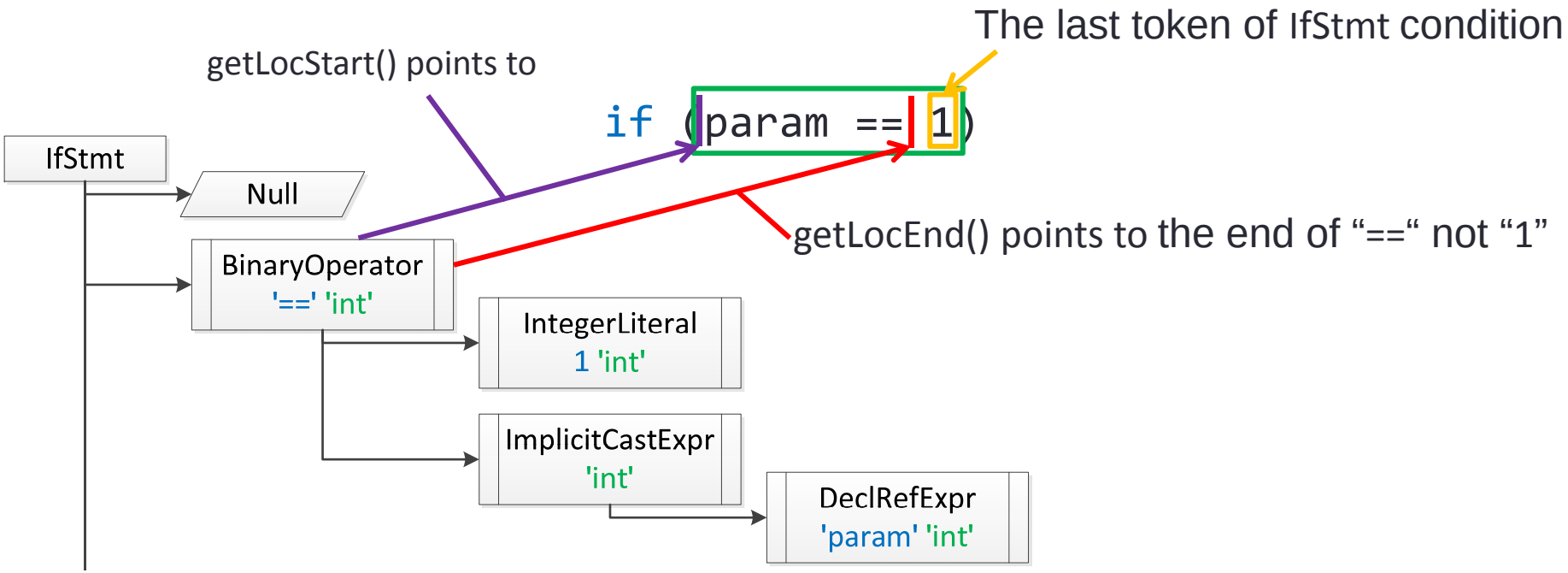
SourceLocation

- To change code, you need to specify where to change
  - Rewriter class requires a `SourceLocation` class instance which contains location information

- You can get a `SourceLocation` instance by:
  - `getLocStart()` and `getLocEnd()` of `Stmt` which return a start and an end locations of `Stmt` instance respectively
  - `findLocationAfterToken(loc, tok,… )` of `Lexer` which returns the location of the first token tok occurring right after loc
    - `Lexer` tokenizes a target code
  - `SourceLocation.getLocWithOffset(offset,…)` which returns location adjusted by the given offset

getLocStart() and getLocEnd()

- getLocStart() returns the exact starting location of Stmt
- getLocEnd() returns the location of Stmt that corresponds to the last-1 th token's ending location of Stmt
  - To get correct end location, you need to use Lexer class in addition
- Example: getLocStart() and getLocEnd() results of IfStmt condition



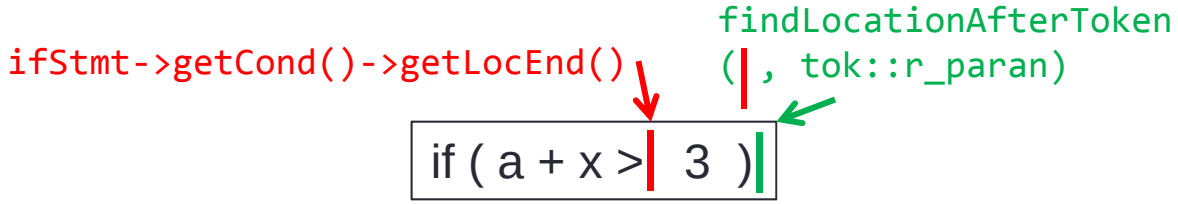
findLocationAfterToken (1/2)

- Static function `findLocationAfterToken(loc,Tkind,…)` of Lexer returns the ending location of the first token of Tkind type after loc

```
static SourceLocation findLocationAfterToken (SourceLocation loc, tok::TokenKind TKind, const
SourceManager &SM, const LangOptions &LangOpts, bool SkipTrailingWhitespaceAndNewLine)
```

- Use `findLocationAfterToken` to get a correct end location of Stmt
  - Example: finding a location of ‘)’ (`tok::r_paren`) using `findLocationAfterToken()` to find the end of if condition

```
1 bool MyASTVisitor::VisitStmt(Stmt *s) {
2   if (isa<IfStmt>(s)) {
3     IfStmt *ifStmt = cast<IfStmt>(s);
4     condition = ifStmt->getCond();
5     SourceLocation endOfCond = clang::Lexer::findLocationAfterToken(condition->
  getLocEnd(), tok::r_paren, m_sourceManager, m_langOptions, false);
6     // endOfCond points ‘)’
7   }
8 }
```



`findLocationAfterToken` (2/2)

- You may find a location of other tokens by changing `TKind` parameter
  - List of useful enums for HW #3

Enum name	Token character
tok::semi	;
tok::r_paren	)
tok::question	?
tok::r_brace	}

- The fourth parameter `LangOptions` instance is obtained from `getLangOpts()` of `CompilerInstance` (see line 99 and line 106 of the appendix)
  - You can find `CompilerInstance` instance in the initialization part of Clang

References

- Clang, http://clang.llvm.org/
- Clang API Documentation, http://clang.llvm.org/doxygen/
- How to parse C programs with clang: A tutorial in 9 parts, http://amnoid.de/tmp/clangtut/tut.html

Appendix: Example Source Code (1/4)

- This program prints the name of declared functions and the class name of each Stmt in function bodies

PrintFunctions.c

```
1  #include <cstdio>
2  #include <string>
3  #include <iostream>
4  #include <sstream>
5  #include <map>
6  #include <utility>
7
8  #include "clang/AST/ASTConsumer.h"
9  #include "clang/AST/RecursiveASTVisitor.h"
10 #include "clang/Basic/Diagnostic.h"
11 #include "clang/Basic/FileManager.h"
12 #include "clang/Basic/SourceManager.h"
13 #include "clang/Basic/TargetOptions.h"
14 #include "clang/Basic/TargetInfo.h"
15 #include "clang/Frontend/CompilerInstance.h"
16 #include "clang/Lex/Preprocessor.h"
17 #include "clang/Parse/ParseAST.h"
18 #include "clang/Rewrite/Core/Rewriter.h"
19 #include "clang/Rewrite/Frontend/Rewriters.h"
20 #include "llvm/Support/Host.h"
21 #include "llvm/Support/raw_ostream.h"
22
23 using namespace clang;
24 using namespace std;
25
26 class MyASTVisitor : public RecursiveASTVisitor<MyASTVisitor>
27 {
28 public:
```

Appendix: Example Source Code (2/4)

```
29      bool VisitStmt(Stmt *s) {
30          // Print name of sub-class of s
31          printf("\t%s \n", s->getStmtClassName() );
32          return true;
33      }
34
35      bool VisitFunctionDecl(FunctionDecl *f) {
36                              // Print function name
37          printf("%s\n", f->getName());
38          return true;
39      }
40  };
41
42  class MyASTConsumer : public ASTConsumer
43  {
44  public:
45      MyASTConsumer()
46      : Visitor() //initialize MyASTVisitor
47      {}
48
49      virtual bool HandleTopLevelDecl(DeclGroupRef DR) {
50          for (DeclGroupRef::iterator b = DR.begin(), e = DR.end(); b != e; ++b) {
51              // Travel each function declaration using MyASTVisitor
52              Visitor.TraverseDecl(*b);
53                              }
54          return true;
55      }
56
57  private:
58      MyASTVisitor Visitor;
59  };
60
61
62  int main(int argc, char *argv[])
63  {
```

Appendix: Example Source Code (3/4)

```
64      if (argc != 2) {
65          llvm::errs() << "Usage: PrintFunctions <filename>\n";
66          return 1;
67      }
68
69      // CompilerInstance will hold the instance of the Clang compiler for us,
70      // managing the various objects needed to run the compiler.
71      CompilerInstance TheCompInst;
72
73      // Diagnostics manage problems and issues in compile
74      TheCompInst.createDiagnostics(NULL, false);
75
76      // Set target platform options
77      // Initialize target info with the default triple for our platform.
78      TargetOptions *TO = new TargetOptions();
79      TO->Triple = llvm::sys::getDefaultTargetTriple();
80      TargetInfo *TI = TargetInfo::CreateTargetInfo(TheCompInst.getDiagnostics(), TO);
81      TheCompInst.setTarget(TI);
82
83      // FileManager supports for file system lookup, file system caching, and directory search management.
84      TheCompInst.createFileManager();
85      FileManager &FileMgr = TheCompInst.getFileManager();
86
87      // SourceManager handles loading and caching of source files into memory.
88      TheCompInst.createSourceManager(FileMgr);
89      SourceManager &SourceMgr = TheCompInst.getSourceManager();
90
91      // Prreprocessor runs within a single source file
92      TheCompInst.createPreprocessor();
93
94      // ASTContext holds long-lived AST nodes (such as types and decls) .
95      TheCompInst.createASTContext();
96
97      // A Rewriter helps us manage the code rewriting task.
98      Rewriter TheRewriter;
```

Appendix: Example Source Code (4/4)

```
 99      TheRewriter.setSourceMgr(SourceMgr, TheCompInst.getLangOpts());
100
101      // Set the main file handled by the source manager to the input file.
102      const FileEntry *FileIn = FileMgr.getFile(argv[1]);
103      SourceMgr.createMainFileID(FileIn);
104
105      // Inform Diagnostics that processing of a source file is beginning.
106      TheCompInst.getDiagnosticClient().BeginSourceFile(TheCompInst.getLangOpts(),&TheCompInst.getPreprocessor());
107
108      // Create an AST consumer instance which is going to get called by ParseAST.
109      MyASTConsumer TheConsumer;
110
111      // Parse the file to AST, registering our consumer as the AST consumer.
112      ParseAST(TheCompInst.getPreprocessor(), &TheConsumer, TheCompInst.getASTContext());
113
114      return 0;
115  }
```