Compilation Principle
编 译 原 理

第16讲：语义分析(4)

张献伟

xianweiz.github.io

DCS290, 4/28/2022

Review Questions

- SDD vs. SDT

  SDD = Syntax Directed Definitions, SDT = SD Translation Schemes

- What are S-SDD and L-SDD?

  S-SDD: synthesized-SDD (only syn attributes),

  L-SDD: left-attributed SDD (only left-to-right dependency).

- Why S-SDD is natural to be implemented in LR parsing?

  Syn attributes: evaluate parent after seeing all children (=reduce).

- Why L-SDD is not natural for LR parsing?

  Semantic actions can be in anywhere of the production body.

- At high level, why L-SDD can be implemented in LR?

  Left-attributed, the needed attribute values must be in the stack.

- Roughly, how do we modify L-SDD for LR parsing?

  Add non-terminal markers to make all actions at production end.

Overview of Symbol Table[符号表]

- **Symbol table** records info of each symbol name in a program[符号表记录每个符号的信息]
  - symbol = name = identifier
- Symbol table is created in the semantic analysis phase[语义分析阶段创建]
  - Because it is not until the semantic analysis phase that enough info is known about a name to describe it
- But, many compilers set up a table at lexical analysis time for the various variables in the program[词法分析阶段准备]
  - And fill in info about the symbol later during semantic analysis when more information about the variable is known
- Symbol table is used in code generation to output assembler directives of the appropriate size and type[后续代码生成阶段使用]

Binding[绑定]

- **Binding**: match identifier use with definition[使用-定义]
  - Definition: associating an *id* with a memory location
  - Hence, binding associates an *id* use with a location
  - Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

Binding[绑定]

- **Binding**: match identifier use with definition[使用-定义]
  - Definition: associating an *id* with a memory location
  - Hence, binding associates an *id* use with a location
  - Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()
{
    char x;    /* allocated at mem[0x100] */
    ...
    {
        int x;  /* allocated at mem[0x200] */
        …
    }

}
```

Binding[绑定]

- **Binding**: match identifier use with definition[使用-定义]
  - Definition: associating an *id* with a memory location
  - Hence, binding associates an *id* use with a location
  - Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()
{
    char x;    /* allocated at mem[0x100] */
    ...
    {
        int x;  /* allocated at mem[0x200] */
        ...
    }
    x = x + 1;
}
```

Binding[绑定]

- **Binding**: match identifier use with definition[使用-定义]
  - Definition: associating an *id* with a memory location
  - Hence, binding associates an *id* use with a location
  - Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()
{
    char x;    /* allocated at mem[0x100] */
    ...
    {
        int x;   /* allocated at mem[0x200] */
        ...
    }
    x = x + 1; /* add mem[0x100],1 ? add mem[0x200],1 ? */
}
```

Binding[绑定]

- **Binding**: match identifier use with definition[使用-定义]
  - Definition: associating an *id* with a memory location
  - Hence, binding associates an *id* use with a location
  - Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()
{
    char x;    /* allocated at mem[0x100] */
    ...
    {
        int x;  /* allocated at mem[0x200] */
        ...
    }
    x = x + 1; /* add mem[0x100],1 ? add mem[0x200],1 ? */
}
```

Binding[绑定]

- **Binding**: match identifier use with definition[使用-定义]
  - Definition: associating an *id* with a memory location
  - Hence, binding associates an *id* use with a location
  - Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()
{
    char x;     /* allocated at mem[0x100] */
    ...
    {
        int x;  /* allocated at mem[0x200] */
        ...
    }
    x = x + 1; /* add mem[0x100],1 ? add mem[0x200],1 ? */
}
```

Scope[作用域]

- **Scope**: program region where a definition can be bound
  - Uses of identifier in the scope is bound to that definition
  - For C: auto/local, static, global

- Some properties of scopes
  - Use not in scope of any definition results in undefined error
  - Scopes for the same identifier can never overlap
    - There is at most one binding at any given time

- Two types: static scoping and dynamic scoping
  - Depending on how scopes are formed

Static Scoping[静态作用域]

- Scopes formed by where definitions are in program text[声明起作用的那段区域]
  - Also known as lexical scoping since related to program text C/C++, Java, Python, JavaScript[也叫词法作用域]
- Rule: bind to the closest enclosing definition

```
void foo()
{
    char x;
    ...
    {
        int x;
        ...
    }
    x = x + 1;
}
```

Dynamic Scoping[动态作用域]

- Scopes formed by when definitions happen during runtime[运行时决定]
  - Perl, Bash, LISP, Scheme
- Rule: bind to most recent definition in current execution

```
void foo()
{
   (1) char x;
   (2) if (...) {
   (3)     int x;
   (4)     ...
         }
   (5) x = x + 1;
 }
```

- Which *x*'s definition is the most recent?
  - Execution (a): ...(1)...(2)...(5)
  - Execution (b): ...(1)...(2)...(3)...(4)...(5)

Static vs. Dynamic Scoping[对比]

- Most languages that started with dynamic scoping (LISP, Scheme, Perl) added static scoping afterwards
- Why? With dynamic scoping ...
  - All bindings are done at execution time
  - Hard to figure out by eyeballing, for both compiler and human
- Pros of static scoping[静态的好处]
  - Static scoping leads to fewer programmer errors
    - Bindings readily apparent from lexical structure of code
  - Static scoping leads to more efficient code
    - Compiler can determine bindings at compile time
    - Compiler can translate identifier directly to memory location
    - Results in generation of efficient code
- We will discuss static scoping only

What is Symbol Table[符号表]

- **Symbol**: same thing as **identifier** (used interchangeably)
- **Symbol table**: a compiler data structure that tracks info about all program symbols
  - Each entry represents a definition of that identifier
  - Maintains list of definitions that reach current program point
  - List updated whenever scopes are entered or exited
  - Used to perform binding of identifier uses at current point
  - Built by either...
    - Traversing the parse tree in a separate pass after parsing
    - Using semantic actions as an integral part of parsing pass
- Usually discarded after generating executable binary
  - Machine code instructions no longer contain symbols
  - For use in debuggers, symbol tables may be included
    - To display symbol names instead of addresses in debuggers
    - For GCC, using ‘gcc -g ...” includes debug symbol tables

Maintaining Symbol Table[维护]

- Basic idea

  int x=0; … void foo() { int x=0; … x=x+1; } … x=x+1 …

  - Start processing *foo*:
    - Add definition of x, overriding old definition of x if any
  - After processing *foo*:
    - Remove definition of x, restoring old definition of x if any
- Operations
  - enter_scope() start a new scope
  - exit_scope() exit current scope

  - find_symbol(*x*) find the information about *x*
  - add_symbol(*x*) add a symbol *x* to the symbol table
  - check_symbol(*x*) true if *x* is defined in current scope

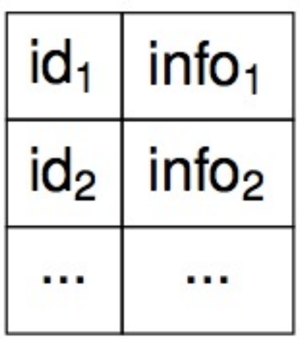
Symbol Table Structure[结构]

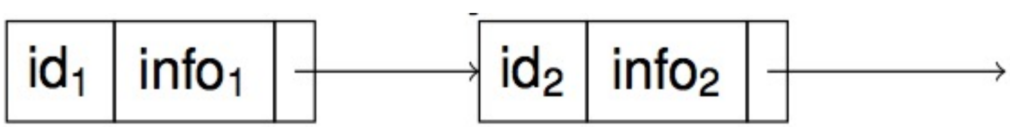
- Frontend time affected by symbol table access time[符号表访问时间影响编译前端性能]
  - Frontend: lexical, syntax, semantic analyses
  - Frequent searches on any large data structure is expensive
  - Symbol table design is important for compiler performance

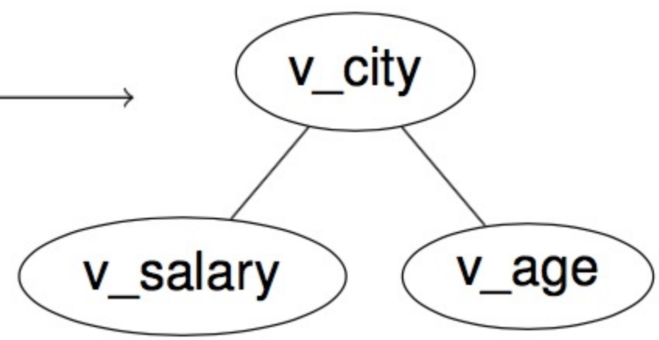
- What data structure to choose?[可选数据结构]
  - **List**[线性表]
  - **Binary tree**[二叉树]
  - **Hash table**[哈希表]
- Tradeoffs: time vs. space[空间和时间的权衡]
  - Let us first consider the organization w/o scope

Symbol Table Structure (cont.)

- **Array**: no space wasted, insert/delete: , search:
- **Linked list**: extra pointer space, insert/delete: , search:
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
- **Binary tree**: use more space than array/list
  - But insert/delete/search is on balanced tree
  - In the worst case, tree may reduce to linked list
    - Then insert/delete/search becomes O(n)

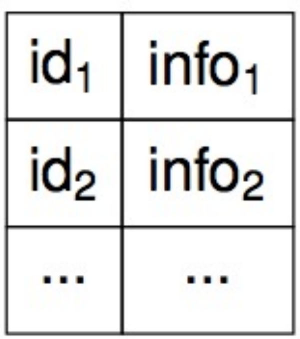


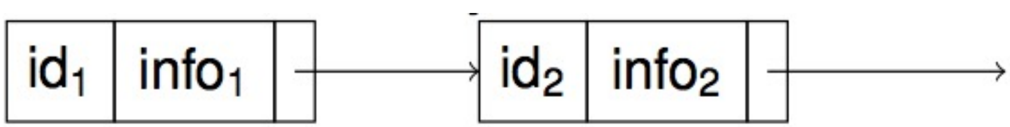


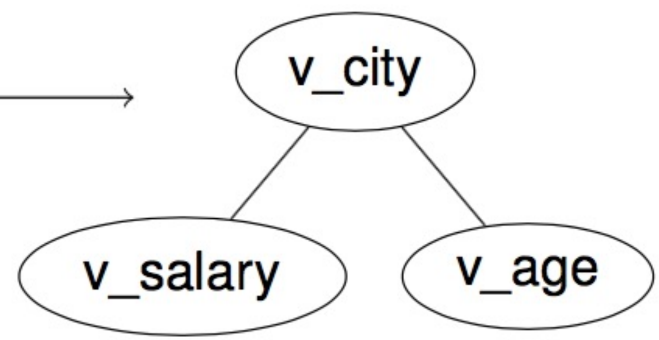


Symbol Table Structure (cont.)

- **Array**: no space wasted, insert/delete: O(n), search:
- **Linked list**: extra pointer space, insert/delete: ,
  search:
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
- **Binary tree**: use more space than array/list
  - But insert/delete/search is on balanced tree
  - In the worst case, tree may reduce to linked list
    - Then insert/delete/search becomes O(n)

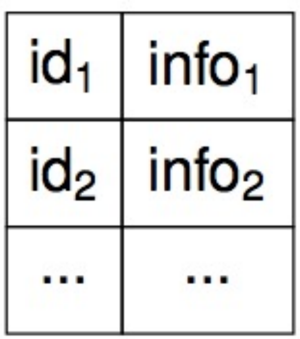


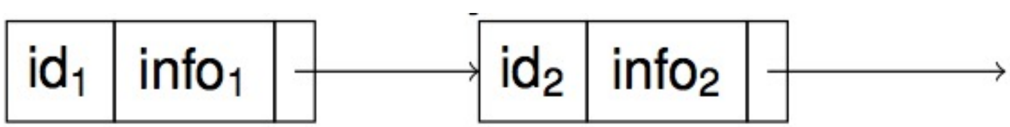


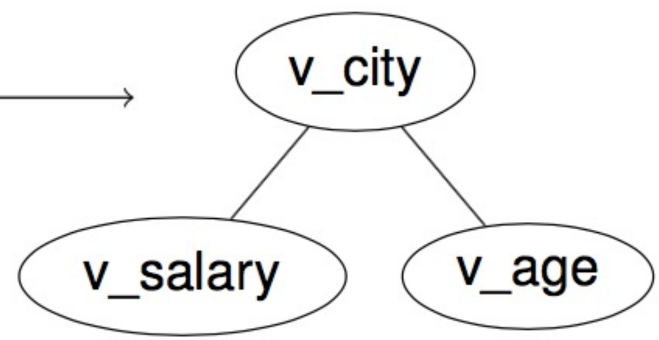


Symbol Table Structure (cont.)

- **Array**: no space wasted, insert/delete: O(n), search: O(n)
- **Linked list**: extra pointer space, insert/delete: ,
  search:
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
- **Binary tree**: use more space than array/list
  - But insert/delete/search is on balanced tree
  - In the worst case, tree may reduce to linked list
    - Then insert/delete/search becomes O(n)

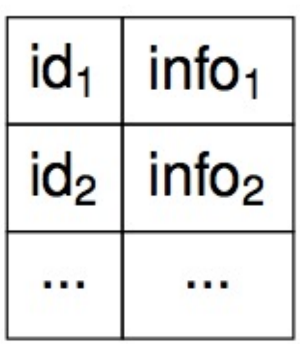


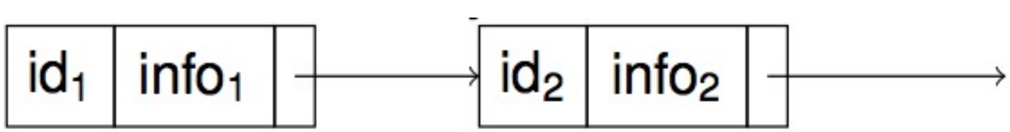


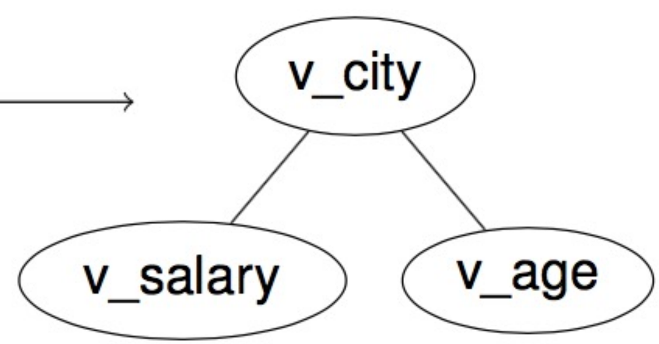


Symbol Table Structure (cont.)

- **Array**: no space wasted, insert/delete: O(n), search: O(n)
- **Linked list**: extra pointer space, insert/delete: O(1) , search:
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
- **Binary tree**: use more space than array/list
  - But insert/delete/search is on balanced tree
  - In the worst case, tree may reduce to linked list
    - Then insert/delete/search becomes O(n)

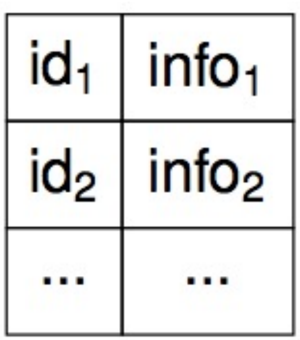


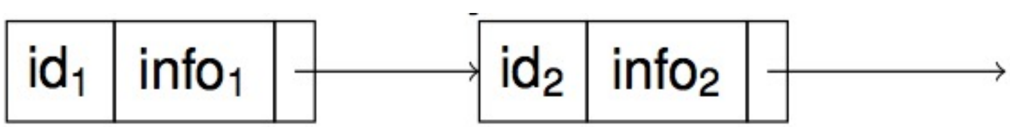


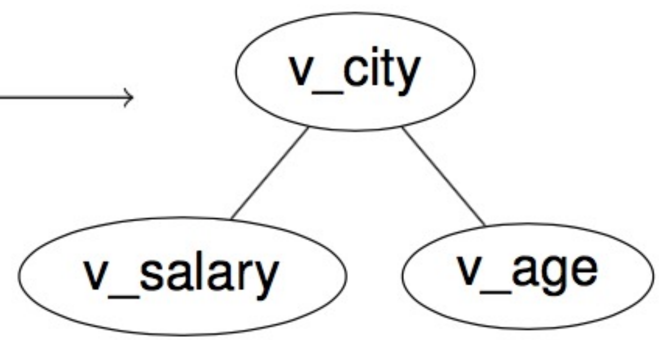


Symbol Table Structure (cont.)

- **Array**: no space wasted, insert/delete: O(n), search: O(n)
- **Linked list**: extra pointer space, insert/delete: O(1), search: O(n)
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
- **Binary tree**: use more space than array/list
  - But insert/delete/search is on balanced tree
  - In the worst case, tree may reduce to linked list
    - Then insert/delete/search becomes O(n)



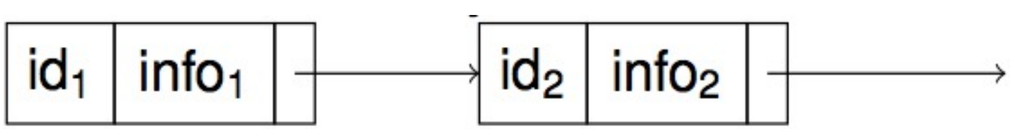


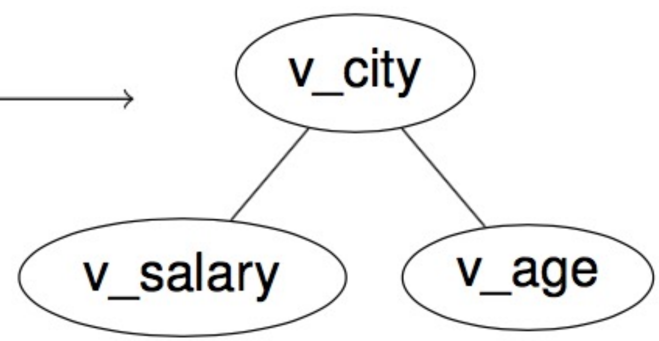


Symbol Table Structure (cont.)

- **Array**: no space wasted, insert/delete: O(n), search: O(n)
- **Linked list**: extra pointer space, insert/delete: O(1) , search: O(n)
  - Optimization: move recently used identifier to the head
  - Frequently used identifiers are found more quickly
- **Binary tree**: use more space than array/list
  - But insert/delete/search is O(log n) on balanced tree
  - In the worst case, tree may reduce to linked list
    - Then insert/delete/search becomes O(n)

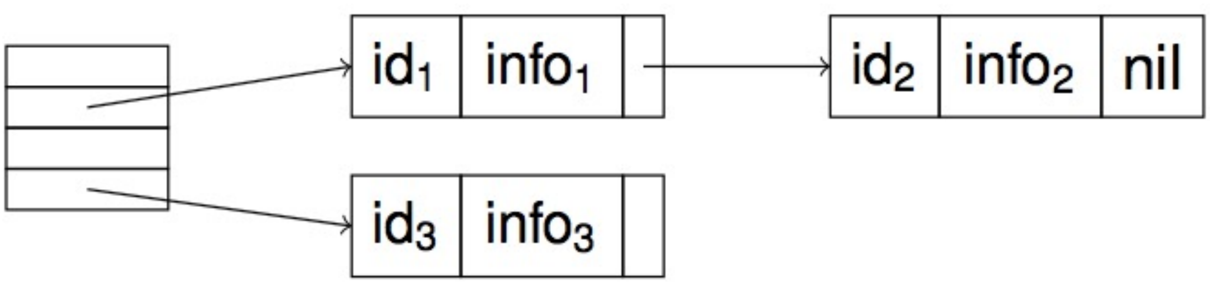
id_1	$info_1$
id_2	$info_2$
...	...





Symbol Table Structure (cont.)

- *hash(id_name)* → *index*[哈希表]
  - A hash function decides mapping from identifier to index
  - Conflicts resolved by chaining multiple IDs to same index
- Memory consumption from hash table (*N* << *M*)
  - M: the size of hash table
  - N: the number of stored identifiers
- But insert/delete/search in O(1) time
  - Can become O(n) with frequent conflicts and long chains
- Most compilers choose hash table for its quick access time



Adding Scope to Symbol Table[作用域]

- To handle multiple scopes in a program,[处理多个作用域]
  - Conceptually, need an individual table for each scope
    - In order to be able to enter and exit scopes
- Sometimes symbols in scope can be discarded on exit:

```
if (...) { int v; } /* block scope */
/* v is no longer valid */
```

- Sometimes not:

Adding Scope to Symbol Table[作用域]

- To handle multiple scopes in a program,[处理多个作用域]
  - Conceptually, need an individual table for each scope
    - In order to be able to enter and exit scopes
- Sometimes symbols in scope can be discarded on exit:

```
if (...) { int v; } /* block scope */
/* v is no longer valid */
```

- Sometimes not:

```
class X { ... void foo() {...} ... } /* class scope */
/* foo() is no longer valid */
X v;
call v.foo(); /* v.foo() is still valid */
```

SUN YAT-SEN UNIVERSITY 中山大學

Adding Scope to Symbol Table[作用域]

- To handle multiple scopes in a program,[处理多个作用域]
  - Conceptually, need an individual table for each scope
    - In order to be able to enter and exit scopes
- Sometimes symbols in scope can be discarded on exit:

```
if (...) { int v; } /* block scope */
/* v is no longer valid */
```

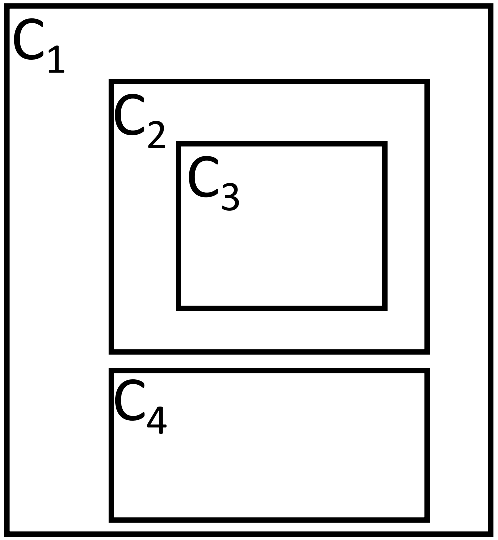
- Sometimes not:

```
class X { ... void foo() {...} ... } /* class scope */
/* foo() is no longer valid */
X v;
call v.foo(); /* v.foo() is still valid */
```

- How can scoping be enforced without discarding symbols?
  - Keep a *stack* of active scopes at a given point
  - Keep a *list* of all reachable scopes in the entire program

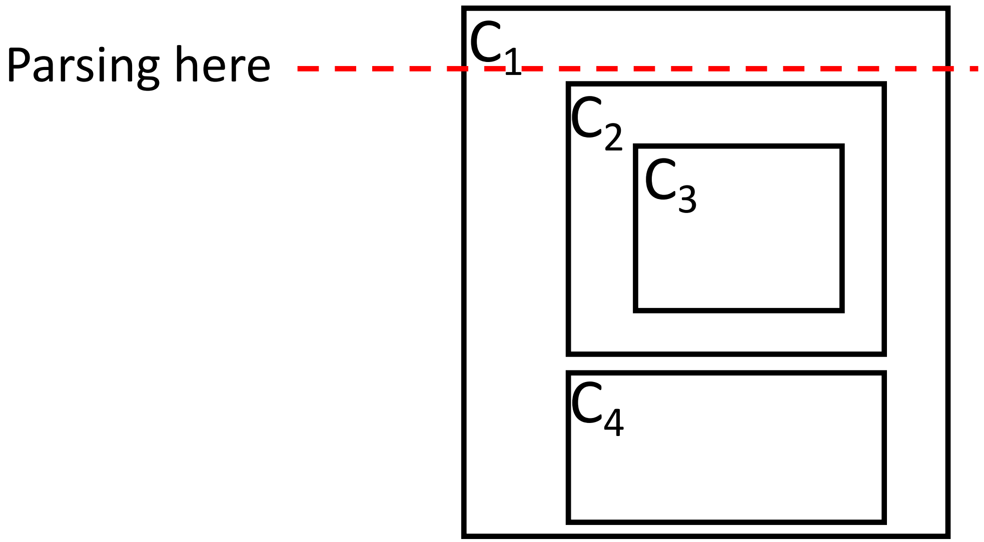
Handle Scopes with Stack

- Organize all symbol tables into a scope stack[作用域栈]
  - An individual symbol table for each scope
    - Scope is defined by nested lexical structure, e.g., {C_1 {C_2 {C_3}} {C_4}}
  - Stack holds one entry for each open scope
    - Innermost scope is stored at the top of the stack
- Stack push/pop happen when entering/exiting a scope



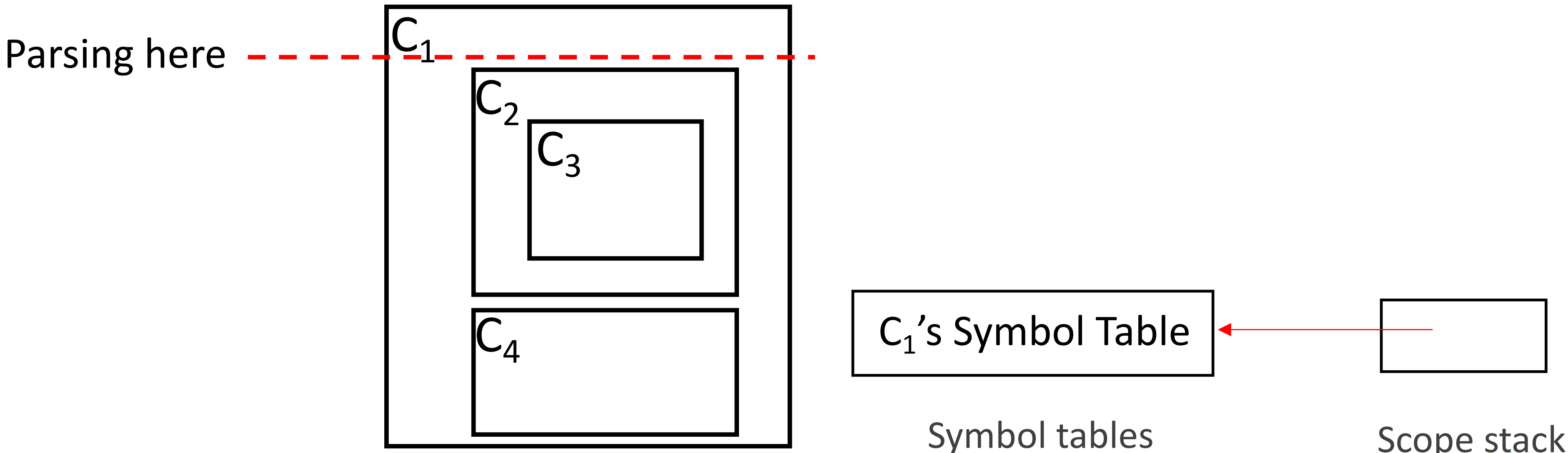
Handle Scopes with Stack

- Organize all symbol tables into a scope stack[作用域栈]
  - An individual symbol table for each scope
    - Scope is defined by nested lexical structure, e.g., {C_1 {C_2 {C_3}} {C_4}}
  - Stack holds one entry for each open scope
    - Innermost scope is stored at the top of the stack
- Stack push/pop happen when entering/exiting a scope



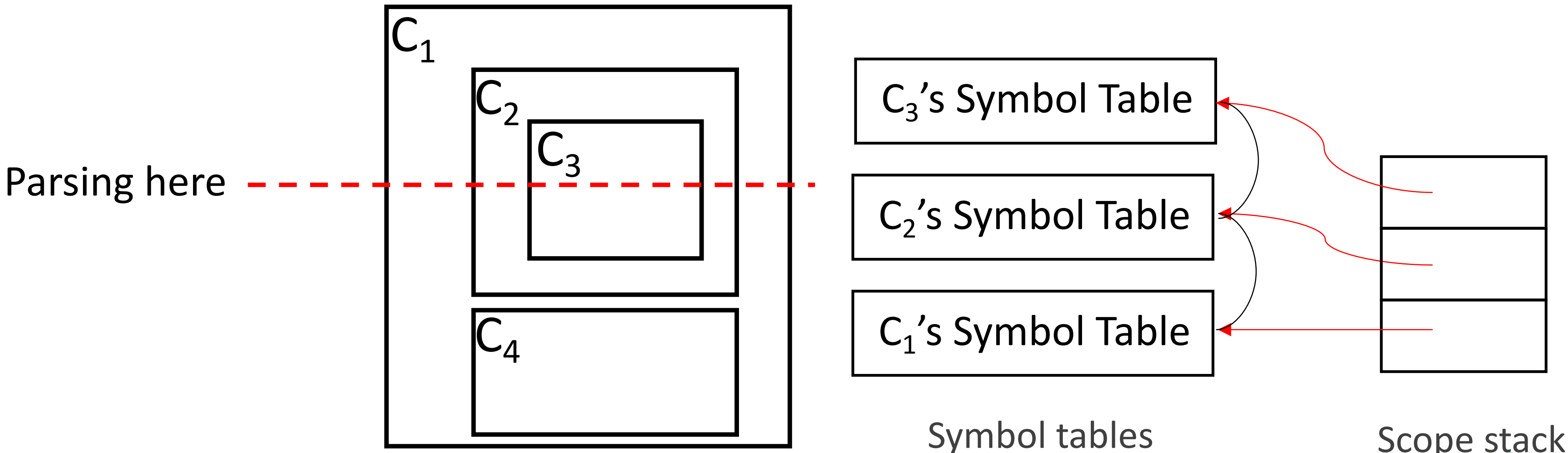
Handle Scopes with Stack

- Organize all symbol tables into a scope stack[作用域栈]
  - An individual symbol table for each scope
    - Scope is defined by nested lexical structure, e.g., {C_1 {C_2 {C_3}} {C_4}}
  - Stack holds one entry for each open scope
    - Innermost scope is stored at the top of the stack
- Stack push/pop happen when entering/exiting a scope



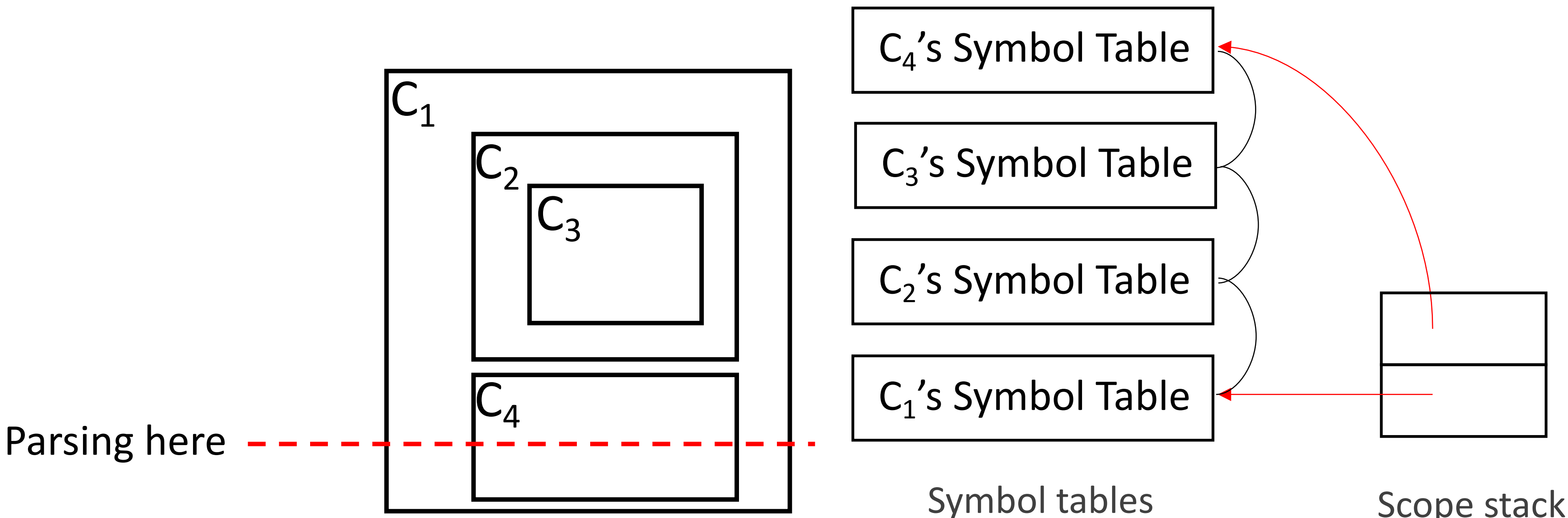
Handle Scopes with Stack (cont.)

- Operations
  - When entering a scope
    - Create a new symbol table to hold all variables declared in that scope
    - Push a pointer to the symbol table on the stack
  - Pop the pointer to the symbol table when exiting scope
  - Search from the top of the stack



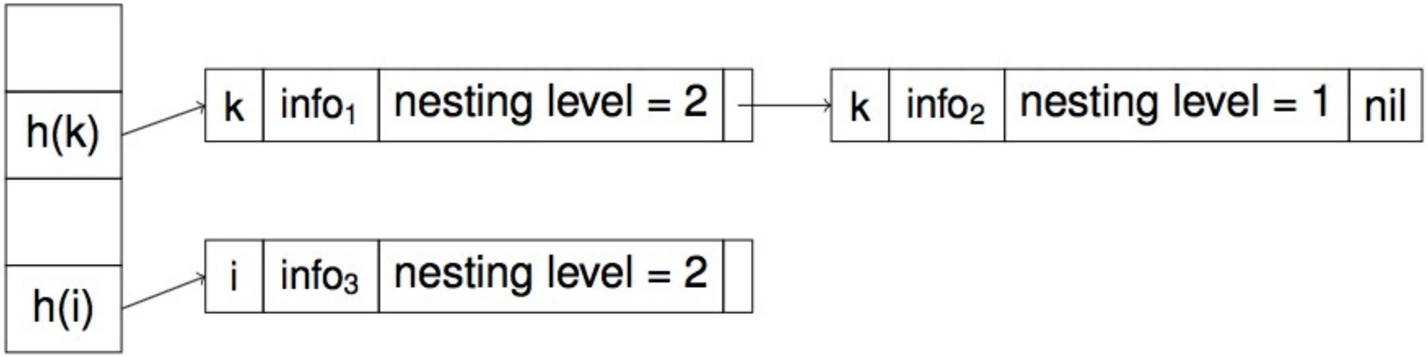
Handle Scopes with Stack (cont.)

- Operations
  - When entering a scope
    - Create a new symbol table to hold all variables declared in that scope
    - Push a pointer to the symbol table on the stack
  - Pop the pointer to the symbol table when exiting scope
  - Search from the top of the stack



Handle Scopes using Chaining

- Cons of stacking symbol tables[栈方式的缺点]
  - Inefficient searching due to multiple hash table lookups
    - All global variables will be at the bottom of the stack
  - Inefficient use of memory due to multiple hash tables
    - Must size hash tables for max anticipated size of scope
- Solution: <u>single symbol table for all scopes</u> using chaining
  - Insert: insert *(ID, current nesting level)* at front of chain
  - Search: fetch ID at the front of chain
  - Delete: when exiting level *k*, remove all symbols with level *k*
    - For efficient deletion, IDs for each level maintained in a list



Handle Scopes using Chaining (cont.)

- Note: symbol table only maintains currently active scopes
  - All entries with the closing scope are deleted upon exiting
- Note: does not maintain list of all reachable scopes
  - Cannot refer back to old scopes that have been exited
  - Still useful for block scopes that are discarded on exit

- Usages
  - Unsuitable for class scopes (only block scopes)[✗]
  - Exiting scopes is slightly more expensive[✗]
    - Requires traversing the entire symbol table
  - Lookup requires only a single hash table access[✓]
  - Savings in memory due to single large hash table[✓]

Info Stored in Symbol Table

- Entry in symbol table
  - String: the name of identifier
  - Kind: function, variable, struct type, class type

string	kind	attributes

- Attributes vary with the kind of symbols
  - variable: type, address of variable
  - function: prototype, address of function body
  - struct type: field names, field types
  - class type: symbol table for class

Attribute List in Symbol Table

- Type info can be arbitrarily complicated
  - Type can be an array with multiple dimensions

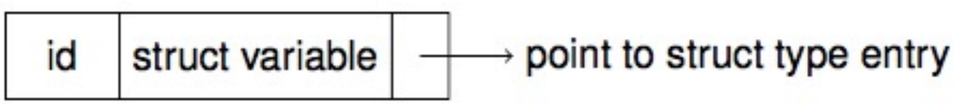
    ```
    char arr[20][20];
    ```

  - Type can be a struct with multiple fields

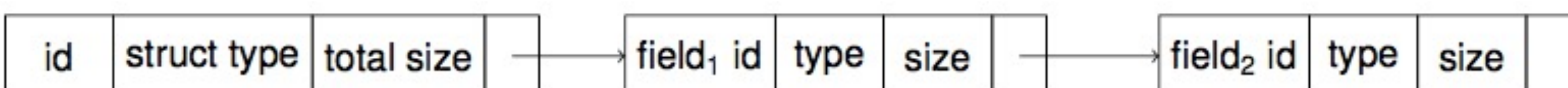
    ```
    struct Point {
        float x;
        float y;
    } point;
    ```

- Store all type info in an attribute list
  - Entry for an array variable with 2 dimensions

  - Entry for a struct variable



  - Entry for a struct type with 2 fields



Use Type Information[类型信息]

- Each variable or function entry contains type info
- Type info is used in later code generation stage[代码生成]
  - To calculate how much memory to alloc for a variable
  - To translate uses of variables to machine instructions
    - Should a ’+’ on variable be an integer or a floating point add?
    - Should a variable assignment be a 4 byte or 8 byte copy?
  - To translate calls to functions to machine instructions
    - What are the types of arguments passed to the function?
    - What is the type of value returned by the function?
- Also used in later code optimization stage[代码优化]
  - To help compiler understand semantics of program
- Also used in semantic analysis stage for ***Type Checking***
  - Uses types to check semantic correctness of program

Type and Type Checking

- **Type**: a set of values + a set of operations on these values
  - int/double: same memory storage
- **Type checking**: verifying type consistency across program[类型一致性检查]
  - A program is said to be <u>type consistent</u> if all operators are consistent with the operand value types
  - Much of what we do in semantic analysis is type checking
- Some type checking examples:
  - Given char *str = “Hello”;
    - str[2] is consistent: char* type allows [] operator
    - str/2 is not: char* type does not allow / operator
  - Given int pi = 3;
    - pi/2 is consistent: int type allows / operator
    - pi=3.14 is not: = operator not allowed on different types
      - Compiler must type convert implicitly to make it consistent

Static Type Checking[静态类型检查]

- Static type checking: at compile time[静态：编译时]
  - Infers program is type consistent through code analysis
    - Collect info via declarations and store in symbol table
    - Check the types involved in each operation
  - E.g., int a, b, c; a = b + c; can be proven type consistent because the addition of two *int*s is an *int*

- Difficult for a language to only do static type checking
  - Some type errors usually cannot be detected at compile time
    - E.g., a and b are of type *int*, a * b may not in the valid range of *int*
    - Typecasting can be pretty risky thing to do (Basically, typecast suspends type checking)
      - unsigned a; (int)a;

Dynamic Type Checking[动态检查]

- Dynamic type checking: at execution time[动态：执行时]
  - Type consistency by checking types of runtime values
  - Include type info for each data location at runtime
    - E.g., a variable of type double would contain both the actual double value and some kind of tag indicating “double type”
    - The execution of any operation begins by first checking these type tags
    - The operation is performed only if everything checks out (otherwise, a type error occurs and usually halts execution)
  - E.g., C++/Java downcasting to a subclass
    - Is `dynamic_cast<Child*>(parent);` type consistent?
  - Array bounds check:
    - Is `int A[10], i; … A[i] = i;` type consistent?
- Static type checking is always more desirable. Why?
  - Always good to catch more errors before runtime
  - Dynamic type checking carries runtime overhead

Static vs. Dynamic Typing[静态-动态]

- Static typing: C/C++, Java, …
  - Variables have static types → hold only one type of value
    - E.g. int x; → x can only hold ints
    - E.g. char *x; → x can only hold char pointers
  - How are types assigned to variables?
    - C/C++, Java: types are explicitly defined
    - int x; → explicit assignment of type int to x
- Pros / cons of static typing
  - More programmer effort
    - Programmer must adhere to strict type rules
    - Defining advanced types can be quite complex (e.g. classes)
  - Less program bugs and execution time
    - Thanks to static type checking

Static vs. Dynamic Typing (cont.)

- Dynamic Typing: Python, JavaScript, PHP, ...
  - Variables have dynamic types → can hold multiple types

```
var x; /* var declaration without a static type */
x = 1; /* now x holds an integer value */
x = "one"; /* now x holds a string value */
```

  - How are types assigned to variables?
    - Type is a runtime property → type tags stored with values
    - Dynamic type checking must be done during runtime
- Pros / cons of dynamic typing
  - Less programmer effort
    - Flexible type rule means program is more malleable
    - Absence of types / classes declarations means shorter code
    - Makes it suitable for scripting or prototyping languages
  - More program bugs and execution time
    - Due to dynamic type checking

Type System[类型系统]

- Static / dynamic typing are type systems
  - **Type System**: types + type rules of a language
- Static / dynamic type checking are methods
  - Methods to enforce the rules of the given type system
- Static type checking is not used exclusively for static typing
  - Static type checking also used for dynamic typing
  - If certain types can be inferred and checked at compile time
    - Can reduce dynamic type checks inserted into code
- Dynamic type checking is not used only for dynamic typing
  - Some features of statically typed languages require it
    - e.g. downcasting requires type check of object type tag

Type Systems: Soundness, Completeness

- Static type checking through inference
  - Inference: deducing a conclusion[结论] from a set of premises[前提]
  - What are the premises? Type rules in the type system
  - What is the conclusion? Accept / reject after applying rules
- A type system is said to be *Sound[可靠]* if:
  - Only correct programs are accepted
  - Flipside: all incorrect programs are rejected
- A type system is said to be *Complete[完备]* if:
  - All correct programs are accepted
  - Flipside: only incorrect programs are rejected
- A type system strives to be both sound and complete
  - The rules of inference (type rules) should reflect that

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Rules of Inference

- What are rules of inference?
  - Inference rules have the form

    if Precondition is true, then Conclusion is true
  - Below concise notation used to express above statement

    $$\frac{\text{Precondition}}{\text{Conclusion}}$$
  - For example: Given E3 → E1 + E2, a rule may be:

    if E1, E2 are type consistent and int types (Precondition),

    then E3 is type consistent and is an int type (Conclusion)
- Recursive type checking via inference
  - Start from variable and constant types at bottom of tree
    - Serves as initial preconditions for the inference
  - Apply rules on operator nodes while working up the tree
    - Checks type consistency and assigns type to node

LLVM: Semantic Analysis

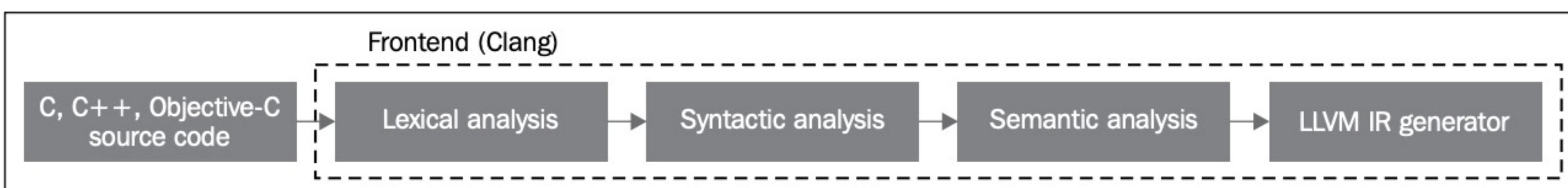
- Clang does not traverse the AST after parsing
  - Instead, it performs type checking on the fly, together with AST node generation

```
StmtResult Parser::ParseIfStatement(SourceLocation *TrailingElseLoc) {

  // perform semantic checking for the if statement, emitting diagnostics accordingly
  return Actions.ActOnIfStmt(IfLoc, IsConstexpr, InitStmt.get(), Cond,
                             ThenStmt.get(), ElseLoc, ElseStmt.get());
}
```

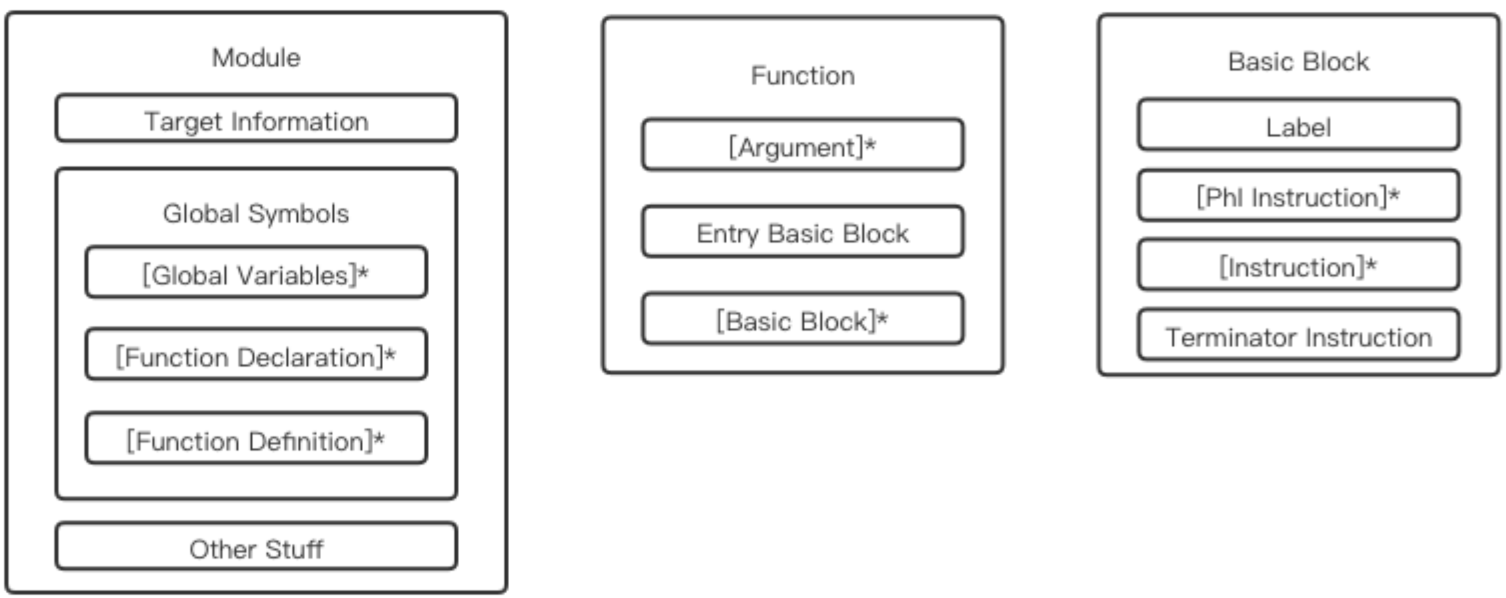
https://github.com/llvm-mirror/clang/blob/master/lib/Parse/ParseStmt.cpp
https://clang.llvm.org/doxygen/ParseAST_8cpp_source.html

  - After the combined parsing and semantic analysis, the ParseAST function invokes the method HandleTranslationUnit to trigger any client that is interested in consuming the final AST.



LLVM: Module

- The Module class represents the top level structure present in LLVM programs
  - An LLVM module is effectively either a translation unit of the original program or a combination of several translation units merged by the linker
  - The Module class keeps track of a list of Functions, a list of GlobalVariables, and a **SymbolTable**



https://releases.llvm.org/7.1.0/docs/ProgrammersManual.html#symboltable

LLVM: Symbol Table

- Public members of Module class
  - *SymbolTable *getSymbolTable()*
    - Return a reference to the SymbolTable for this Module
  - *Function *getOrInsertFunction(const std::string &Name, const FunctionType *T)*
    - Look up the specified function in the Module SymbolTable. If it does not exist, add an external declaration for the function and return it.
  - *std::string getTypeName(const Type *Ty)*
    - If there is at least one entry in the SymbolTable for the specified Type, return it. Otherwise return the empty string
  - *bool addTypeName(const std::string &Name, const Type *Ty)*
    - Insert an entry in the SymbolTable mapping Name to Ty. If there is already an entry for this name, true is returned and the SymbolTable is not modified.