KU | Fall 2018 | Drew Davidson

EECS 665 COMPILER CONSTRUCTION

19 – Name Analysis

Lecture Outline

- Last time
  - Introduce semantic analysis
  - Talk about program semantics / PL design
- This time
  - Continue semantic analysis
  - Consider an instance of semantic analysis: name analysis (AKA name resolution)

Semantic Analysis: Passes Over AST

- Walk the abstract syntax tree
  - Collect and leverage context sensitive information
- Will allow ourselves to walk the completed AST post-construction
  - In contrast to parse tree we will keep around the AST data structure explicitly
- Structured as a series of passes

Semantic Analysis: Implementation

- Some passes performed as traversal of AST
- Requires a repository of context-sensitive information
  - *Symbol Table:* Mapping of information about semantic symbols (i.e. functions, variables, *etc.*)

Symbol Table Entries

- May contain symbol info such as:
  - Kind (e.g., struct, variable, function, class)
  - Type (e.g., int, int x string -> bool, struct)
  - Nesting level
  - Runtime location (where it’s stored in memory)

Symbol Table Operations

- Insert entry
- Lookup
- Add a new “context”
- Remove/forget a “context”

Name Analysis: Definitions

- Associates IDs with their uses in the program
- Prerequisite for type analysis (assigning types to expressions)
  - Need to bind names before we can type uses

Name Analysis: Purpose

- Prerequisite for code generation
- Catch some obvious errors
  - Undeclared identifiers

Scope

- A central issue in name analysis is to determine the **lifetime** of a variable, function, *etc.*
- Scope definition: the block of code in which a name is visible/valid

Scope: Feature of a Language

- Some languages have NO notion of scope
  - Assembly / FORTRAN
- Most familiar: static / most deeply nested
  - C / C++ / Java

There are several decisions to make, we'll overview a couple of them

Forms of Scope

- Scoping is a design decision of the programming language
  - No scope
    - Assembly / FORTRAN
  - Most deeply nested
    - C / C++ / Java

Scope Decision: Variable Shadowing

Variable Shadow:variable shadowing occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope

- Do we allow nested names to be re-used?

- What about when the kinds are different?

```
void smoothJazz(int a){
 int a;
 if (a){
   int a;
   if (a){
     int a;
   }
  }
}

void hardRock(int a){
 int hardRock;
}
```

Scope Decision: Static v Dynamic

- Static Scope
  - Most deeply nested w.r.t. **syntactic** block (determined at compile time)
- Dynamic Scope
  - Most deeply nested w.r.t. **calling context** (determined at runtime)

Static v Dynamic Example

```
int a = 1
int hop(){
     return a;
}
int hip(){
     int a = 2;
     return hip();
}
int hippo(){
     return hip();
}
```

Scope Decision: Overloading

- Do we allow same names, same scope, different types?

```
int techno(int a){ … }
bool techno(int a){ … }
bool techno(bool a){ … }
bool techno(bool a, bool b){ }
```

Scope Decision: Forward Reference

- Do we allow use before name is (lexically) defined?

```
void country() {
    western();
}
void western() {
    country();
}
```

- Requires 2 passes over the program
  - 1 to fill symbol table
  - 1 pass to use symbols

OUR Scope Decisions

- What scoping rules will we employ?
- What info does the compiler need in its symbol table?

Lil' C: A statically scoped language

- Language is designed for ease of symbol table use:
  - Global scope + nested scope
  - All declarations made at the top of a block
  - Variable cannot be referenced after its declared block

Lil’ C: Examples

```
int a;
void func(){
    int b;
    a = 1;
    b = 2;
}
```

```
int a;
void func(){
    a = 1;
    int b;
    b = 2;
}
```

Lil' C: Nesting

- Shadowing:
  - Variable shadowing allowed
  - Struct definition shadowing allowed
- Resolved via most deeply nested lexical scope (aka static scope)

Lil’ C: Nesting Examples

```
int a;
void fun(){
    int a;
    a = 1;
}
```

```
int a;
void fun(){
    a = 1;
}
```

Lil' C: Nesting Examples

```
struct SDef {
  int a;
};
void fun(){
  struct SDef{
    int b;
  };
  struct SDef c;
}
```

```
struct SDef {
  int a;
};
void fun(){
  struct SDef{
    int b;
  };
  struct SDef c;
  c.a = 4;
}
```

Lil’ C: Symbol Table Implementation

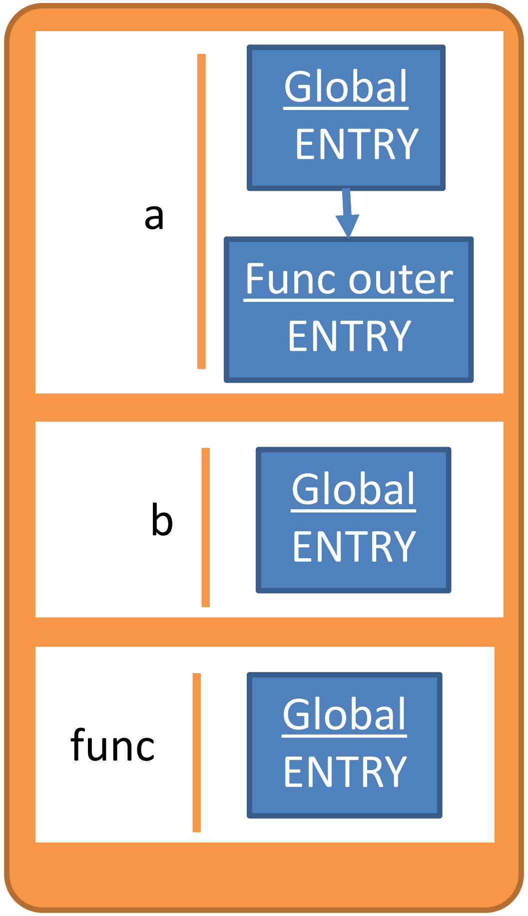
- We want a symbol table to *efficiently* add an entry when we need it, remove it when we’re done
- We’ll go with a list of hashmaps
  – (Worse) alternative : hashmap of lists

Lil' C: Symbol Table Implementation

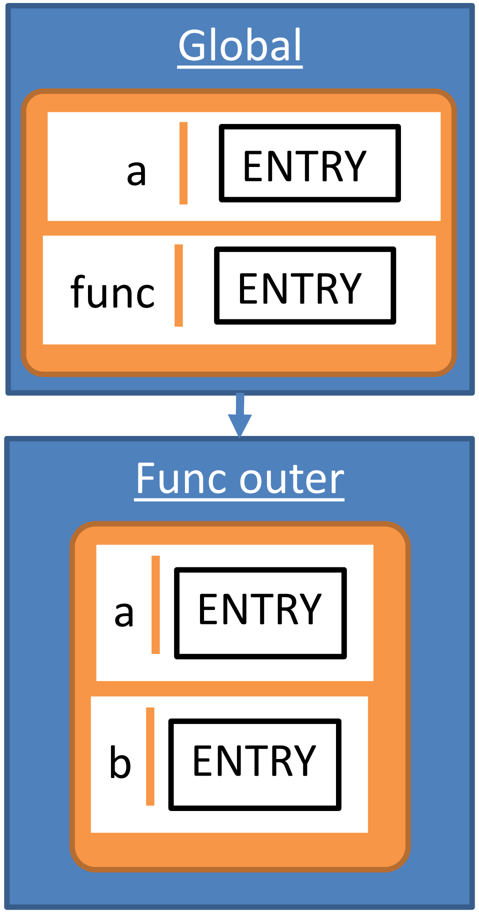
Code

```
int a;
func(){
   int a;
   int b;
}
```

Hashmap of lists



List of hashmaps

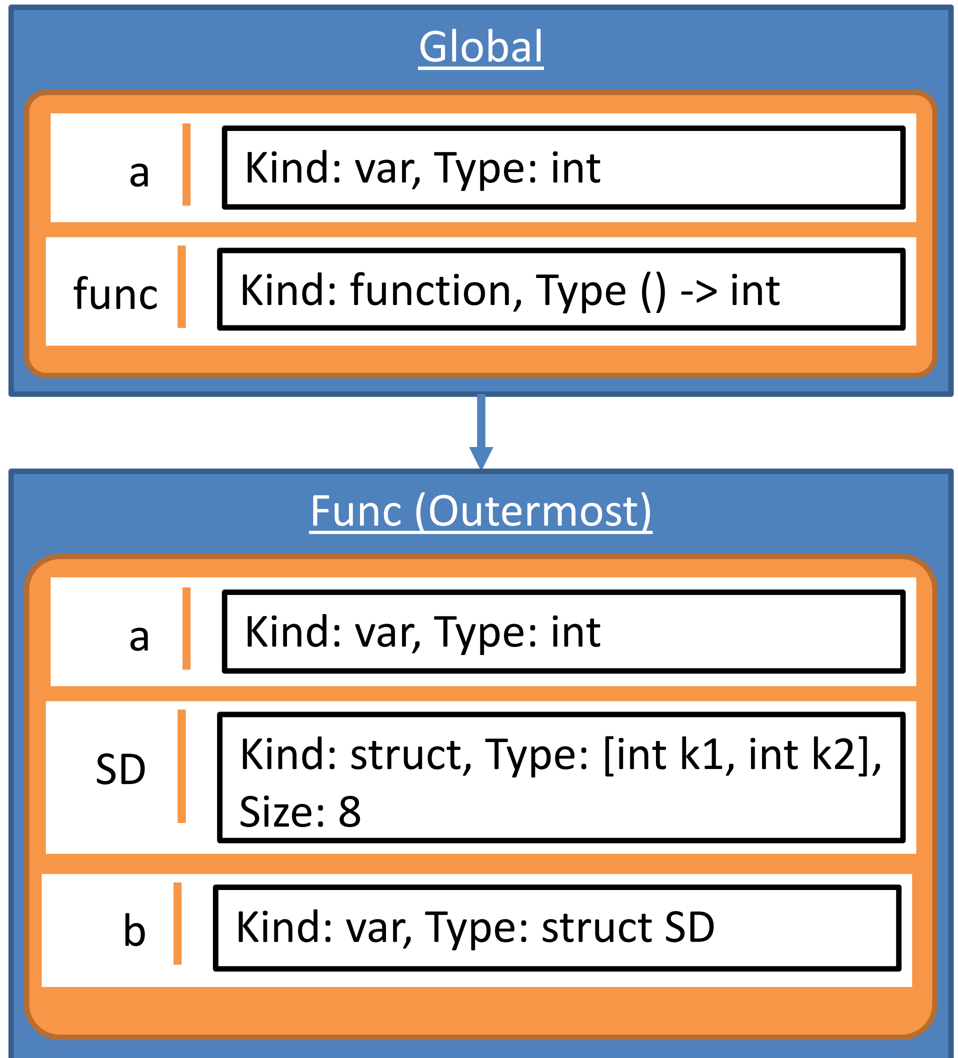


Lil’ C: Symbol Kinds

- Identifier types
  - Variable
    - Carries a name, type
  - Function declarations
    - Carries a name, return type, list of parameter types
  - Struct definitions
    - Carries a name, list of fields (types with names), size

Lil’ C: Symbol Table Implementation

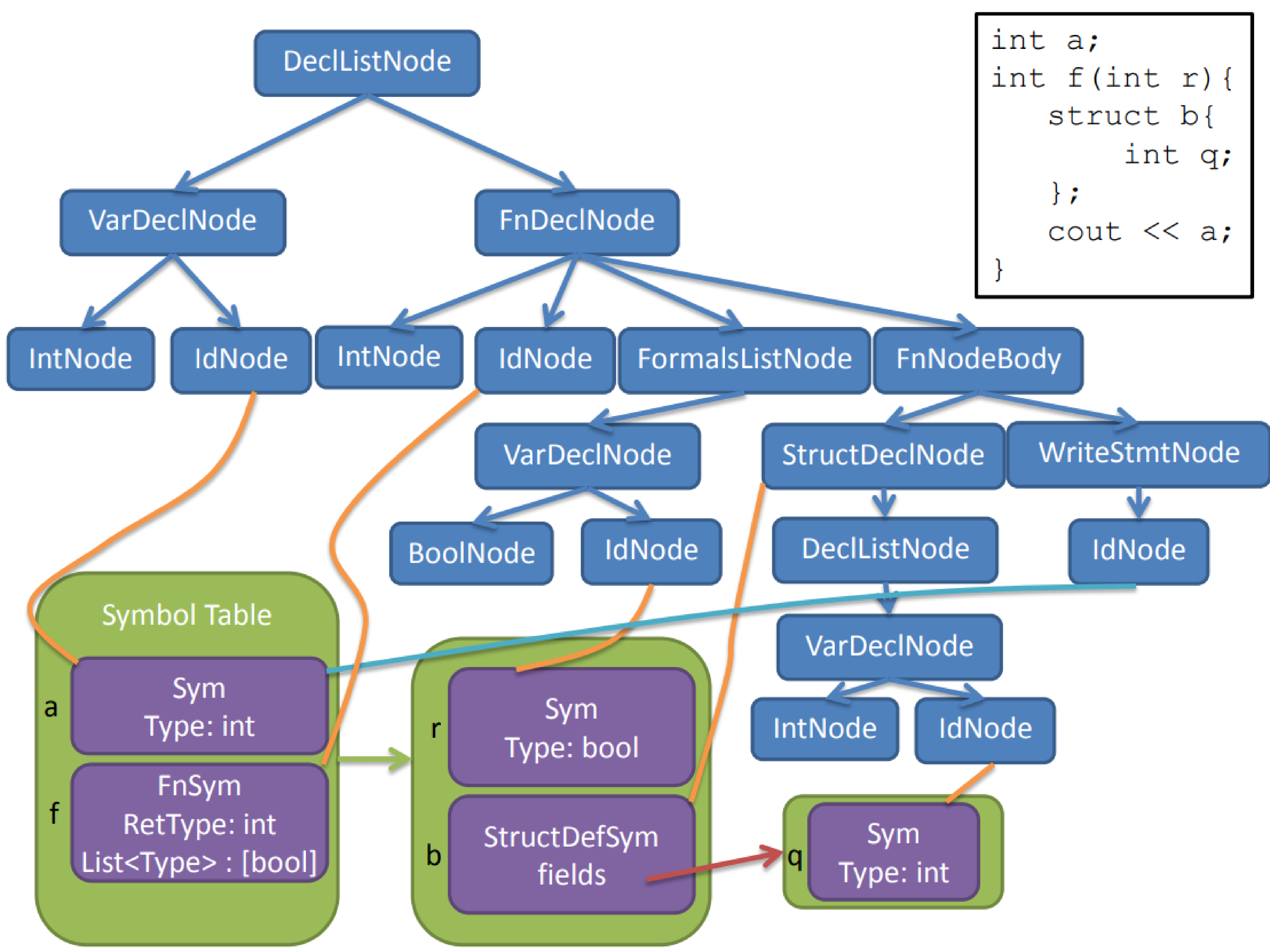
```
int a;
int func(){
   int a;
   struct SD{
      int k1;
      int k2;
   };
   struct SD b;
}
```



Using the Symbol Table

Implementing Name Analysis

- Walk the AST, much like the unparse() method
  - Augment AST nodes with a link to the relevant name in the symbol table
  - Build new entries into the symbol table when a declaration is encountered


DeclListNode
VarDeclNode
FnDeclNode
IntNode
IdNode
IntNode
IdNode
FormalsListNode
FnNodeBody
VarDeclNode
StructDeclNode
WriteStmtNode
BoolNode
IdNode
DeclListNode
IdNode
VarDeclNode
IntNode
IdNode
Symbol Table
a
Sym
Type: int
f
FnSym
RetType: int
List<Type> : [bool]
r
Sym
Type: bool
b
StructDefSym
fields
q
Sym
Type: int
int a;
int f(int r){
struct b{
int q;
};
cout << a;
}

Next Lecture

- Build on our name analysis information to perform a *type* analysis
  - Not only type variables, but also functions and expressions