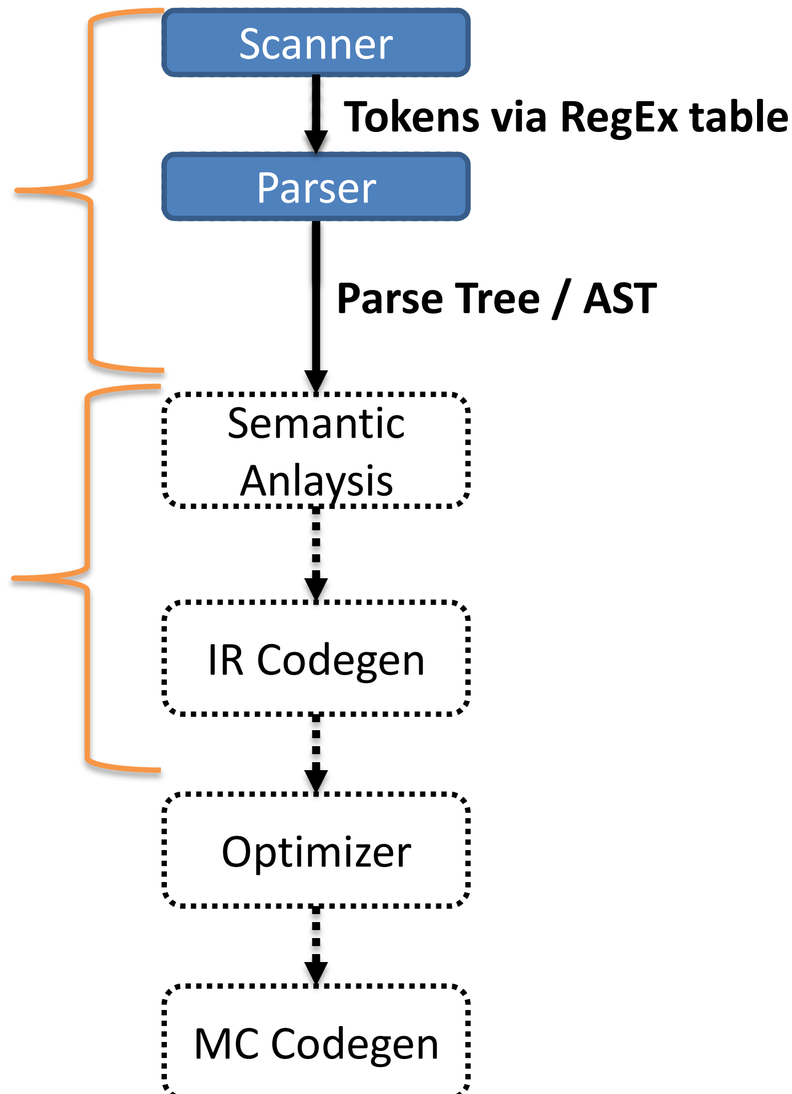
CS536

Semantic Analysis Introduction with Emphasis on Name Analysis

Where we are at

- So far, we've only defined the *structure* of a program: AKA the *syntax*
- We are now diving into the *semantics* of the program



Semantics: The **Meaning** of a Program

- The parser can guarantee that the program is *structurally* correct
- The parser does not guarantee that the program makes sense:

```
void var;
```

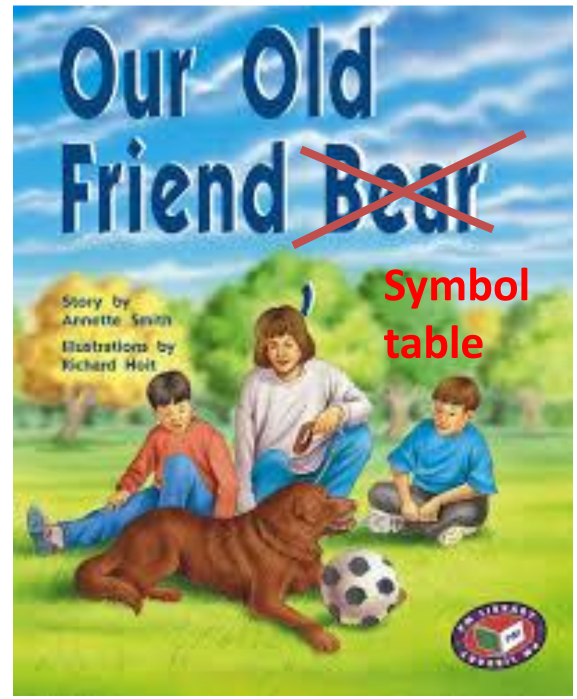
Undeclared variables

Ill-typed statements

```
int doubleRainbow;
doubleRainbow = true;
```

Static Semantic Analysis

- Two phases:
  - Name analysis (aka name resolution)
    - For each scope
      - Process declarations, add them to the symbol table
      - Process statements, update IDs to point to their entry
  - Type analysis
    - Process statements
      - Use symbol table info to determine the type of each expression



Why do we need this phase?

- Code generation
  - Different operations use different instructions:
    - Consistent variable access
    - Integer addition vs fp addition
    - Operator overloading
- Optimization
  - Symbol table knows* where a variable is used
    - Can remove dead code
    - Can weaken the type (e.g. int -> bool)
- Error checking

***pointers can make this occasionally impossible to know**

Semantic Error Analysis

- For non-trivial programming languages, we run into fundamental undecidability problems
  - Halting?
  - Crashes?
- Sometimes practical feasibility as well
  - Thread interleavings
  - Interprocedural dataflow

Catch Obvious Errors

- We may not be able to guarantee the absence of errors
- We can at least catch some, though
  - Undeclared identifiers
  - Multiply declared identifiers
  - Ill-typedness

Name Analysis

- Associating ids with their uses
- Need to bind names before we can type uses
  - What definition information do we need about identifiers?
  - How do we bind definitions and uses together?

In other words, what do we store in the symbol table?

scope

Symbol Table Entries

- A table that binds a name to information we need
- Information typically needed in an entry
  - Kind (struct, variable, function, class)
  - Type (int, int × string → bool, struct)
  - Nesting level
  - Runtime location (where it's stored in memory)

Symbol Table Operations

  - Insert entry
  - Lookup
  - Add new table
  - Remove/forget a table
- When should we use these operations?

Scope: the lifetime of a name

- Block of code in which a name is visible/valid
  - No scope
    - Assembly / FORTRAN
  - static / most deeply nested scope
    - Should be familiar
      - C / Java / C++

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

void soul(int b){
    if (b){
        int c = 2;
    }
}
```

Many decisions related to scope

Static vs Dynamic Scope

- Static
  - Correspondence between a variable use / decl is known at compile time
- Dynamic
  - Correspondence determined at runtime

```
void hip(){
  a = 1;
}

void hop(){
  a = 2;
}

void hippo(){
  a++;
}
```

Variable Shadowing

- Do we allow names to be reused in nesting relations?

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

- What about when the *kinds* are different?

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

Overloading

- Same name, different types

```
int techno(int a){
}

bool techno(int a){
}

bool techno(bool a){
}

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

Forward References

- Use of a name before it is filled out in the symbol table

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

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

- Requires two passes over the program:
  - 1 to fill symbol table, 1 to use it

Scope Exercise

Look at some scope rules in languages we know

Name analysis for C-Flat

- Time to make some decisions
  - What scoping rules will we allow?
  - What info does a C-Flat compiler need in it's symbol table?
  - Relevant for Project 4

C-Flat: A statically scoped language

- C-Flat is designed for ease of symbol table use
  - global scope + nested scopes
  - All declarations are made at the top of a scope
  - Declarations can always be removed from table at end of scope

```
int a;
void fun(){
    int b;
    int c;
    int d;
    b = 0;
    if (b == 0){
        int d;
    }
    c = b;
    d = b + c;
}
```

C-Flat: Nesting

- Like Java or C, we'll use most deeply nested scope to determine binding
  - Shadowing
    - Variable shadowing allowed
    - Struct definition shadowing allowed

```
int a;
void fun(){
    int b;
    b = 0;
    if (b == 0){
        int b;
        b = 1;
    }
    c = b;
}
```

C-Flat: Symbol Table Implementation

- We want the symbol table to *efficiently* add an entry when we need it, remove it when we're done with it
- We'll go with a list of hashmaps
  - This makes sense since we expect to remove a lot of names from scope at once

C-Flat: Symbol Kinds

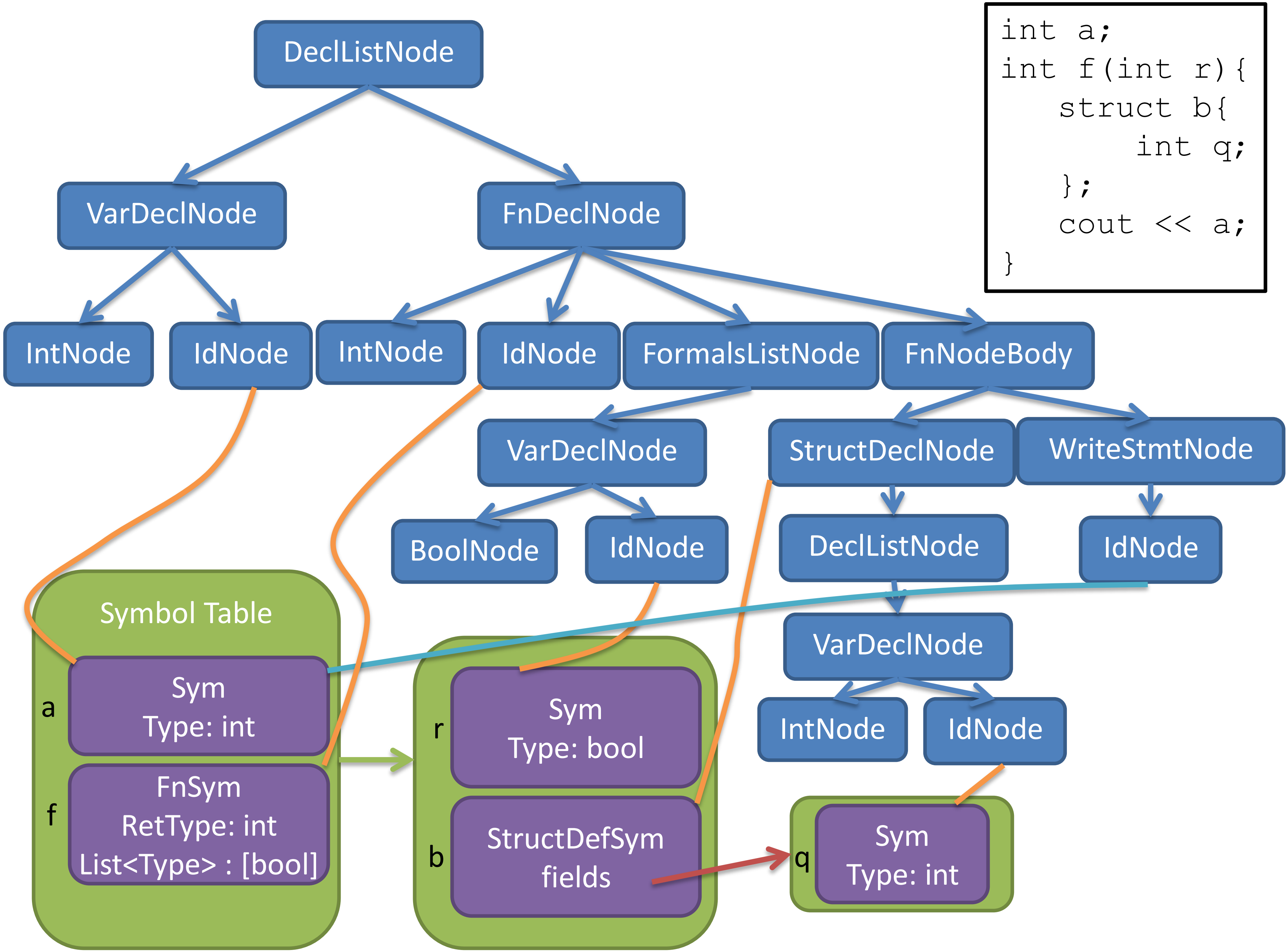
- Identifier types
  - Variables
    - Carries a name, primitive type
  - Function Declarations
    - Carries a name, return type, list of param types
  - Struct Definitions
    - Carries a name, list of fields (types with names), size

C-Flat: Sym Class Implementation

- There are many ways to implement your symbols
- Here’s one suggestion:
  - Sym class for variable definitions
  - FnSym subclass for function declarations
  - StructDefSym for struct *type* definitions
    - Contains it’s OWN symbol table for it’s field definitions
  - StructSym for when you want an instance of a struct

Implementing Name Analysis with an AST

- At this point, we're basically done with the Parse Tree
- 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


int a;
int f(int r){
struct b{
int q;
};
cout << a;
}
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