Announcements

Midterm review this Thursday

Try the midterm I put online

Homework solutions now available

Semantic Analysis with Emphasis on Name Analysis

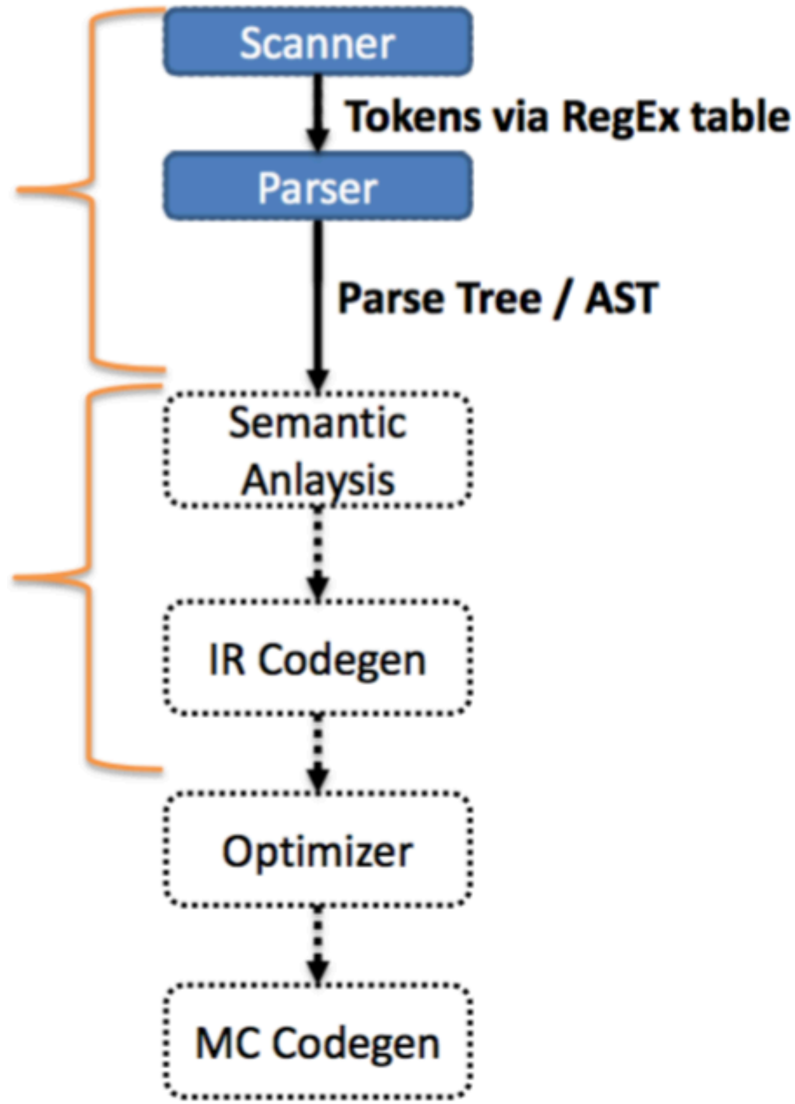
You'll need this for P4

We'll get back to Parsing next week

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 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 (and sub-expression)

Why do we need this phase?

Code generation

- Different operations use different instructions:
  - Consistent variable access
  - Integer addition vs floating point addition
  - Operator overloading

Optimization

- Symbol table knows where a variable is used
  - Can remove dead code
  - Can weaken the type (e.g., int -> bool)
  - NOTE: pointers can make this occasionally impossible

Error checking

Semantic Error Analysis

For non-trivial programming languages, we run into fundamental undecidability problems

- Does the program halt?
- Can the program crash?

Sometimes practical feasibility as well

- Combinations thread interleavings
- Inter-procedural dataflow

Catch Obvious Errors

We cannot guarantee absence of errors…

…but we can at least catch some:

- Undeclared identifiers
- Multiply declared identifiers
- Ill-typed terms

Name analysis

Associating ids with their uses

Need to bind names before we can type uses

- What definitions do we need about identifiers?
  - Symbol table
- How do we bind definitions and uses together?
  - Scope

Symbol table entries

Table that binds a name to information we need

What information do you think we need?

- Kind (struct, variable, function, class)
- Type (int, int × string → bool, struct)
- Nesting level
- Runtime location (where it's stored in memory)

Any name can be put into the table.

Symbol table operations

- Insert entry
- Lookup
- Add new table
- Remove/forget a table

When do you think we use these operations?

Scope: the lifetime of a name

Block of code in which a name is visible/valid

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

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

No scope

- Assembly / FORTRAN

Static / most nested scope

- Should be familiar – C / Java / C++

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 main() {
  f1();
  f2();
}

void f1() {
  int x = 10;
  g();
}

void f2() {
  String x = "hello";
  f3();
  g();
}

void f3() {
  double x = 30.5;
}

void g() {
  print(x);
}
```

Exercises

```
class animal {
  // methods
  void attack(int animal) {
     for (int animal=0; animal<10; animal++) {
         int attack;
     }
  }

  int attack(int x) {
     for (int attack=0; attack<10; attack++) {
        int animal;
     }
  }

  void animal() { }

  // fields
  double attack;
  int attack;
  int animal;
}
```

What uses and declarations are OK in this Java code?

Exercises

```
void main() {
  int x = 0;
  f1();
  g();
  f2();
}

void f1() {
  int x = 10;
  g();
}

void f2() {
  int x = 20;
  f1();
  g();
}

void g() {
  print(x);
}
```

What does this print, assuming dynamic scoping?

Variable shadowing

Do we allow names to be reused in nesting relations?

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;
}
```

Overloading

Same name different type

```
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 added to symbol table

How do we implement it?

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

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

Requires two passes over the program

- 1 to fill symbol table, 1 to use it

Example

```
int k=10, x=20;

void foo(int k) {
    int a = x;
    int x = k;
    int b = x;
    while (...) {
       int x;
       if (x == k) {
          int k, y;
          k = y = x;
       }
       if (x == k) {
          int x = y;
       }
    }
}
```

Determine which uses correspond to which declarations

Example

```
int (1)k=10, (2)x=20;

void (3)foo(int (4)k) {
    int (5)a = x(2);
    int (6)x = k(4);
    int (7)b = x(6);
    while (...) {
       int  (8)x;
       if (x(8) == k(4)) {
          int (9)k, (10)y;
          k(9) = y(10) = x(8);
       }
       if (x(8) == k(4)) {
          int (11)x = y(ERROR);
       }
    }
}
```

Determine which uses correspond to which declarations

Name analysis for our language

Time to make some decisions

- What scoping rules will we allow?
- What info does our project compiler need in its symbol table?
- Relevant for P4

Our language is statically scoped

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;
}
```

Our language: 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;
}
```

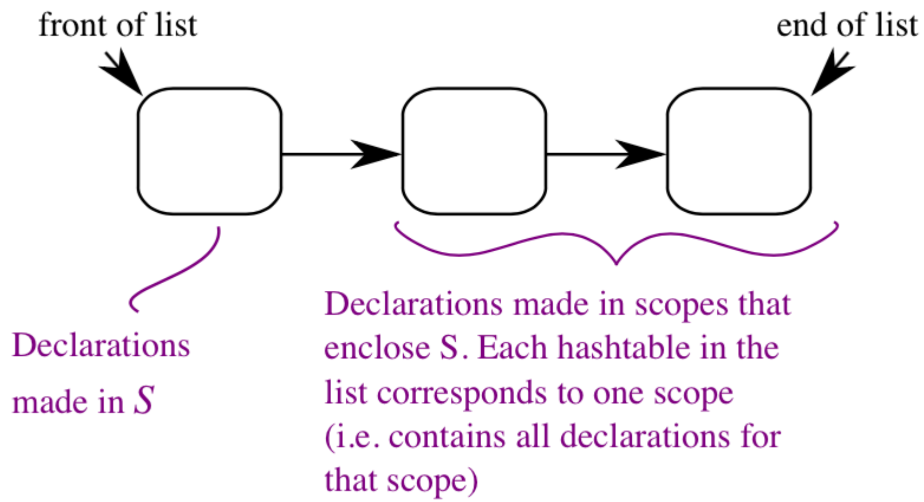
Our language: Symbol table implementation

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

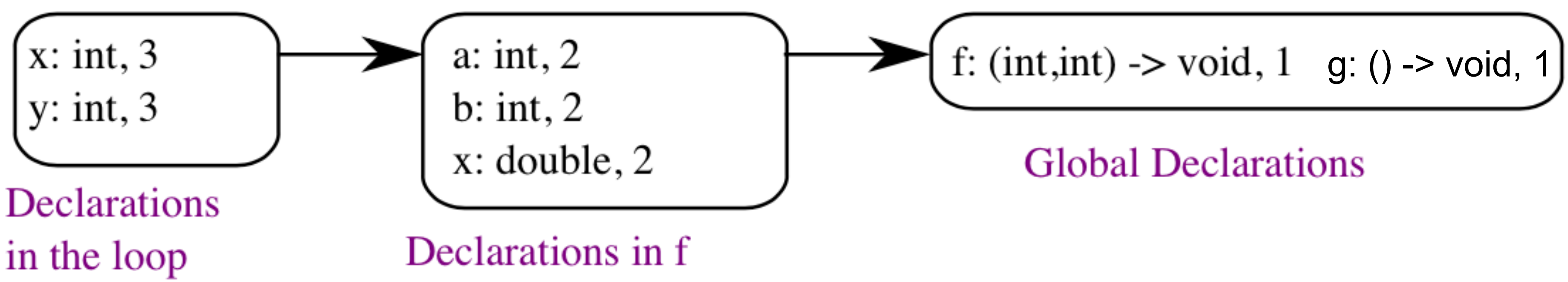
- This makes sense since we expect to remove a lot of names from scope at once
- You did most of this in P1

Example




```
void f(int a, int b) {
   double x;
   while (...) {
      int x, y;
      ...
   }

void g() {
   f();
}
```



Our language: Symbol kinds

Symbol kinds (= types of identifiers)

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

Our language: 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 done with the Parse Tree

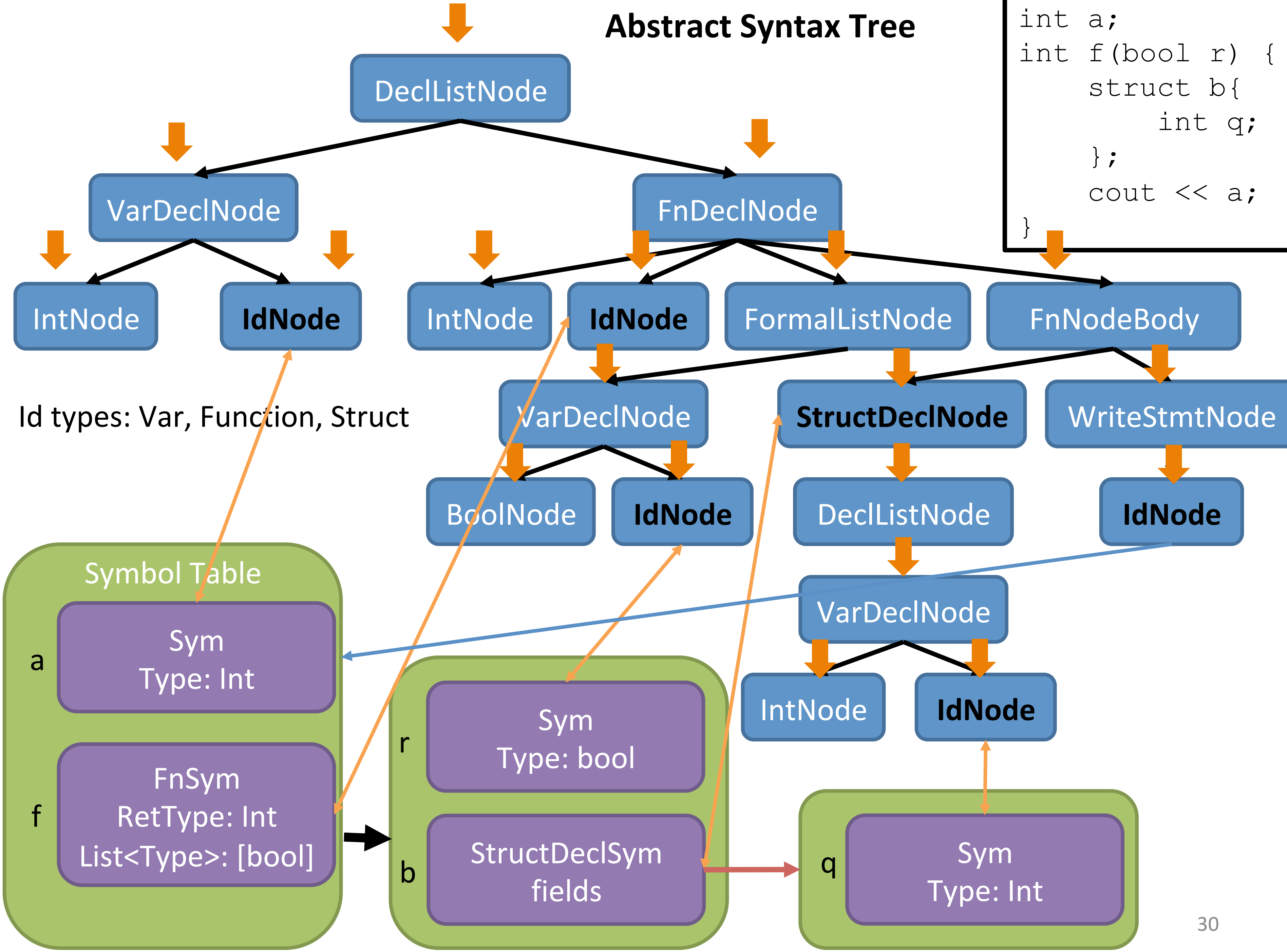
- All subsequent processing done on the AST + symbol table

Walk the AST, much like the unparse() method

- Augment AST nodes where names are used (both declarations and uses) with a link to the relevant object in the symbol table
- Put new entries into the symbol table when a declaration is encountered

Abstract Syntax Tree

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



Id types: Var, Function, Struct