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 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 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
  – 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 definitions do we need about identifiers?
    • Symbol table
  – How do we bind definitions and uses together?
    • 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++

```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
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;
}
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 return, assuming dynamic scoping?
Variable shadowing

• Do we allow names to be reused in nesting relations?
• What about when the kinds are different?

```c
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

```cpp
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

```c
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;
        }
    }
}
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);
        }
    }
}
Name analysis for YES

• Time to make some decisions
  – What scoping rules will we allow?
  – What info does a YES compiler need in its symbol table?
  – Relevant for P4
YES: A statically scoped language

- YES 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

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

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

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

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

void g() {
    f();
}

Declarations made in scopes that enclose S. Each hashtable in the list corresponds to one scope (i.e. contains all declarations for that scope)

Declarations in the loop

Declarations in f
YES: 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
YES: 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;
}