Semantic Analysis
with Emphasis on Name Analysis

You’ll need this for P4
Where we are at

So far, we’ve only defined the structure of a program—a.k.a. 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:
- Undeclared variables
- Ill-typed statements

```cpp
int doubleRainbow;
doubleRainbow = true;
```
Static Semantic Analysis

Two phases

– Name analysis (a.k.a. name resolution)
  • For each scope
    – Process declarations, insert them into the symbol table
    – Process statements, update IdNodes to point to the appropriate symbol-table 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 entry serves to identify which variable is used
  - Can help in removing dead code (with some further analysis)
  - NOTE: pointers can make these tasks hard

Error checking
Semantic Error Analysis

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

• Does the program halt?
• Can the program crash?

Even with simplifying assumptions, sometimes infeasible in practice, as well

• Combinations of thread interleavings
• Inter-procedural dataflow
Catch Obvious Errors

We cannot guarantee the 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

(Structured) dictionary that binds a name to information that 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 is stored in memory)
Symbol-Table Operations

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

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

```java
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);
}
```
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);
}
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 added to symbol table
How do we implement it?

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

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

Requires two passes over the program
– 1 to fill symbol table, 1 to use it
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 b

Time to make some decisions

- What scoping rules will we allow?
- What info does a b compiler need in its symbol table?
- Relevant for P4
b: A Statically Scoped Language

b 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;
}
```
b: Nesting

Like Java or C, we’ll use most deeply nested scope to determine binding

- Shadowing
  - Variable shadowing allowed

```c
int a;
void fun(){
  int b;
  b = 0;
  if (b == 0){
    int b;
    b = 1;
  }
  c = b;
}
```
b: Symbol-Table Implementation

We want a symbol-table implementation for which we can

– add an entry efficiently when we need to
– remove an entry when we are done with it

We will use a list of hashmaps

– sensible because we expect to remove a lot of names from a scope at once
– you did most of this in P1
Example

```java
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)

Global Declarations

```plaintext
x: int, 3
y: int, 3
Declarations in the loop

a: int, 2
b: int, 2
x: double, 2
Declarations in f

f: (int,int) -> void, 1
```
Symbol kinds (= types of identifiers)

- Variable
  - Carries a name, primitive type

- Function declaration
  - Carries a name, return type, list of parameter types

- Struct definition
  - Carries a name, list of fields (types with names), size
b: Implementation of Class Sym

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 its field definitions
– StructSym for when you want an instance of a struct
Implementing Name Analysis with an AST

At this point, we are done with the parse tree (which never existed to begin with 😊)

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