
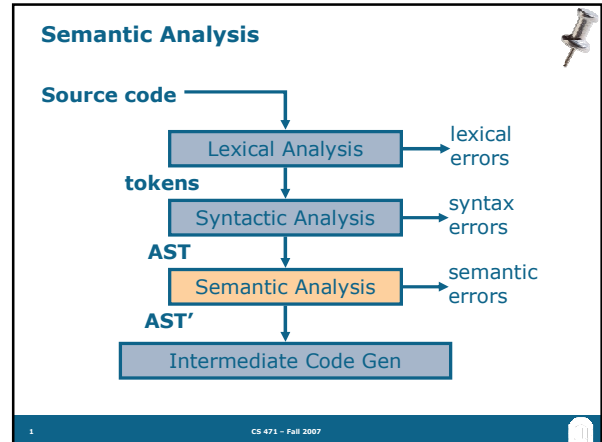


Semantic Analysis: Scope

CS 471
October 1, 2007

The Compiler So Far

Lexical analysis

- Detects inputs with illegal tokens
 - e.g.: main\$ ();

Parsing

- Detects inputs with ill-formed parse trees
 - e.g.: missing semicolons

Semantic analysis

- Last "front end" analysis phase
- Catches all remaining errors

2 CS 471 - Fall 2007

Last Time

- We can build an interpreter and/or typechecker, code generator directly into our YACC specification
- Why and how we generate **Abstract Syntax Trees**
 - In theory *and* in YACC
 - We now have a head start on PA4

```

A_SeqExp(2,
  A_ExpList(A_AssignExp(4,
    A_SimpleVar(2,
      S_Symbol("a")), A_IntExp(7,5)),
  A_ExpList((A_OpExp(11,A_plusOp,
    A_VarExp(A_SimpleVar(10,
      S_Symbol("a"))),A_IntExp(12,1))), NULL)))
  
```

- Now we can *really* move on to **Semantic Analysis**

3 CS 471 - Fall 2007

Goals of a Semantic Analyzer

Compiler must do more than recognize whether a sentence belongs to the language...

- Find all possible remaining errors that would make program invalid
 - undefined variables, types
 - type errors that can be caught **statically**

Terminology

Static checks - done by the compiler
Dynamic checks - done at run time

4 CS 471 - Fall 2007

The Varargs Bug

- A few things still slip by!
- Try compiling this code:

```

void main()
{
    int i=21, j=42;
    printf("Hello World\n");
    printf("Hello World, N=%d\n");
    printf("Hello World\n", i, j);
    printf("Hello World, N=%d\n");
    printf("Hello World, N=%d\n");
}
  
```

5 CS 471 - Fall 2007

Why Separate Semantic Analysis?

Parsing cannot catch some errors
Why?

Some language constructs are not context-free

- Example: All used variables must have been declared (scoping)
- Example: A method must be invoked with arguments of proper type (typing)

6

CS 471 - Fall 2007

What Does Semantic Analysis Do?

Checks of many kinds:

1. All identifiers are declared
 2. Types
 3. Inheritance relationships
 4. Classes defined only once
 5. Methods in a class defined only once
 6. Reserved identifiers are not misused
- And others . . .

The requirements depend on the language

7

CS 471 - Fall 2007

Typical Semantic Errors

• **Multiple declarations:** a variable should be declared (in the same scope) at most once

• **Undeclared variable:** a variable should not be used before being declared

• **Type mismatch:** type of the left-hand side of an assignment should match the type of the right-hand side

• **Wrong arguments:** methods should be called with the right number and types of arguments

8

CS 471 - Fall 2007

A Sample Semantic Analyzer

Works in two phases

- traverses the AST created by the parser

1. For each scope in the program

- **process the declarations**
 - add new entries to the symbol table and
 - report any variables that are multiply declared
- **process the statements**
 - find uses of undeclared variables, and
 - update the "ID" nodes of the AST to point to the appropriate symbol-table entry.

2. Process all of the statements in the program again

- use the symbol-table information to determine the type of each expression, and to find type errors.

9

CS 471 - Fall 2007

Scoping

In most languages, the same name can be declared multiple times

- if its declarations occur in different scopes, and/or
- involve different kinds of names

Java: can use same name for

- a class
- field of the class
- a method of the class
- a local variable of the method

```
class Test {  
    int Test;  
    void Test( ) { double Test; }  
}
```

10

CS 471 - Fall 2007

Scoping: Overloading

Java and C++ (but not in Pascal or C):

- can use the same name for more than one method
- as long as the number and/or types of parameters are unique

```
int add(int a, int b);  
float add(float a, float b);
```

11

CS 471 - Fall 2007

Scoping: General Rules

The scope rules of a language:

- Determine which declaration of a named object corresponds to each use of the object
- Scoping rules map uses of objects to their declarations

C++ and Java use *static scoping*:

- Mapping from uses to declarations at compile time
- C++ uses the "most closely nested" rule
 - a use of variable **x** matches the declaration in the most closely enclosing scope
 - such that the declaration precedes the use

12

CS 471 - Fall 2007

Scope levels

Each function has two or more scopes:

- One for the function body
 - Sometimes parameters are separate scope!
 - (Not true in C)

```
void f( int k ) { // k is a parameter
  int k = 0; // also a local variable
  while (k) {
    int k = 1; // another local variable, in a loop
  }
}
```

- Additional scopes in the function
 - for each *for* loop and
 - each nested block (delimited by curly braces)

13

CS 471 - Fall 2007

POP QUIZ

- Match each use to its declaration, or say why it is a use of an undeclared variable.

```
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; }
  }
}
```

14

CS 471 - Fall 2007

Dynamic scoping

Not all languages use static scoping

Lisp, APL, and Snobol use *dynamic scoping*

Dynamic scoping:

- A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the **most-recently-called still active** function

15

CS 471 - Fall 2007

Example

For example, consider the following code:

```
int i = 1;
void func() {
  cout << i << endl;
}
int main() {
  int i = 2;
  func();
  return 0;
}
```

If C++ used dynamic scoping, this would print out 2, not 1

16

CS 471 - Fall 2007

Pop Quiz #2

- Assuming that dynamic scoping is used, what is output by the following program?

```
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); }
```

17

CS 471 - Fall 2007

Variables/Identifiers

Need an environment that keeps track of types of all identifiers in scope

```

{
  int i, n = ...;
  for (i=0; i < n; i++)
    boolean b = ...
}

```

18 CS 471 – Fall 2007

Symbol Tables

purpose:

- keep track of names declared in the program
- names of
 - variables, classes, fields, methods

symbol table entry:

- associates a name with a set of attributes, e.g.:
 - kind of name (variable, class, field, method, etc)
 - type (int, float, etc)
 - nesting level
 - memory location (i.e., where will it be found at runtime)

19 CS 471 – Fall 2007

Symbol Tables

- Symbol table** (also called environments)
- Can be represented as set of name \rightarrow type pairs (**bindings**) {a \rightarrow string, b \rightarrow int}

Functions:
 Type Lookup(String id)
 Void Add(String id, Type binding)

20 CS 471 – Fall 2007

Environments

Represents a set of mappings in the symbol table

```

function f(a:int, b:int, c:int) =
  ( print_int(a+c);
    let var j := a+b
      var a := "hello"
      in print(a); print_int(j)
    end;
    print_int(b)
  )

```

Lookup in σ_1

σ_0
 $\sigma_1 = \sigma_0 + a \rightarrow \text{int}$
 $\sigma_2 = \sigma_1 + j \rightarrow \text{int}$
 σ_1
 σ_0

21 CS 471 – Fall 2007

Imperative vs. Functional Environments

Functional style – keep all copies of $\sigma_0 \sigma_1 \sigma_2 \dots$
Imperative style – modify σ_1 until it becomes σ_2

- “destroys” σ_1
- can “undo” σ_2 to get back to σ_1 again
- single environment σ that becomes $\sigma_1 \sigma_2 \sigma_3$
- latest bindings destroyed, old bindings restored

NOTE: Functional/imperative environment management can be used regardless of whether the language is “functional” “imperative” or “object-oriented”

How would you implement an imperative environment?

22 CS 471 – Fall 2007

Implementation options

- Hash tables (pointing to linked lists)**
- Binary search trees**

- We’ll talk more about the implementation (in theory and in your Tiger compiler) next week
- For now, we’ll continue with a theoretical view of the second task of semantic analysis: typing
 - (Covered only indirectly in the book!)

23 CS 471 – Fall 2007

Summary

- **Semantic analysis**

- last analysis of "front end"
- Checks for "context-sensitive errors"

- **Scoping**

- Static or dynamic

