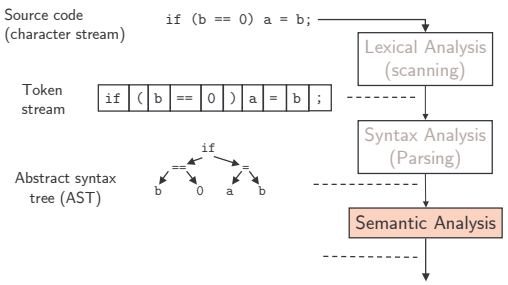
CS412/413

Introduction to Compilers

Radu Rugina

Lecture 11: Symbol Tables
15 Feb 06

Where we Are



Incorrect Programs

- Programs with correct syntax may still contain errors
- Lexical and syntax analysis are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc.
- Example: lexical analysis does not distinguish between different variable names; it returns the same ID token

```
int a;          int a;
a = 1;          b = 1;
```


Incorrect Programs

- Example: syntax analysis does not correlate variable declarations with variable uses:

```
int a;
a = 1;          a = 1;
```

- Example: syntax analysis does not keep track of types:

```
int a;          int a;
a = 1;          a = 1.0;
```


Goals of Semantic Analysis

- Semantic analysis = ensure that the program satisfies a set of rules regarding the usage of programming constructs (e.g., variables, objects, expressions, statements)
- Examples of semantic rules:
  - A variable should not be defined multiple times
  - Variable must be declared before being used
  - Variables must be defined before being used
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The test an if statement must have boolean type
- Typing rules are an important class

Type Information

- Type information = describes what values correspond can program constructs have: variables, statements, expressions, functions, etc.

variables:	`int i;`	integer
expressions:	`(i+1) == 2`	boolean
statements:	`while(i<5) i++;`	void
functions:	`int pow(int n,int m)`	int x int → int

- Type checking = set of rules which ensures the type consistency of different constructs in the program
  - Will discuss it in more detail next two lectures

Scope and Visibility

- Scope (or visibility) of an identifier = the portion of the program where the identifier can be referred to
- Lexical scope = textual region in the program
  - Statement block
  - Formal argument list
  - Object body
  - Function or method body
  - Module or file
  - Whole program (multiple modules)
- Scope of an identifier: the lexical scope its declaration refers to

Scope and Visibility

- Scope of variables in statement blocks:

```
{ int a;
  ...
     { int b;
     }
  ...
}
```

scope of variable a

scope of variable b

- Global variables in C:
  - If declared "static" then the current file
  - If declared "extern" then the whole program

Scope and Visibility

- Scope of formal arguments of functions/methods:

```
int factorial(int n) {
   ...
}
```

scope of argument n

- Scope of labels in C:

```
void f() {
   ... goto l; ...
   l: a =1;
   ... goto l; ...
}
```

scope of label l

Scope and Visibility

- Scope of object fields and methods:

```
class A {
   private int x;
   public void g() { x=1; }
   ...
}

class B extends A {
   ...
   public int h() { g(); }
   ...
}
```

scope of field x

scope of method g

Declarations

- Usually, identifiers must be declared in their scopes
  Rule 1: Use an identifier only if declared in enclosing scope
  Rule 2: Do not declare identifiers of the same kind with identical names more than once in the same lexical scope
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds

```
class X {
   int X;
   void X(int X) {
      X: for(;;)
         break X;
   }
}
```

```
int X(int X) {
  int X;
  goto X;
    { int X;
      X: X = 1; }
}
```

Not Recommended!

Symbol Tables

- Symbol table = an environment that stores information about identifiers
  - It is an important data structure, used throughout the rest of the compilation process
- Each entry in the symbol table contains
  - The name of an identifier
  - Attributes: its kind, its type, type qualifiers, etc.

NAME	KIND	TYPE	QUALIFIER
foo	fun	int x int → bool	extern
m	param	int	
n	param	int	const
tmp	var	bool	const

Scope Information

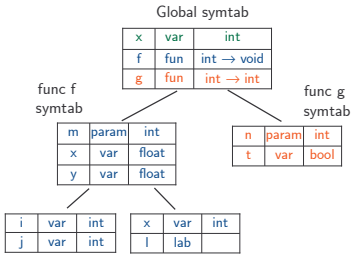
- How to capture the scope information in the symbol table?
- Idea:
  - There is a hierarchy of scopes in the program
  - Use a similar hierarchy of symbol tables
  - One symbol table for each scope
  - Each symbol table contains the symbols declared in that lexical scope

Example

```
int x;

void f(int m) {
    float x, y;
    ...
   { int i, j; ...; }
   { int x; l: ...; }
}

int g(int n) {
    bool t;
    ...;
}
```



Identifiers With Same Name

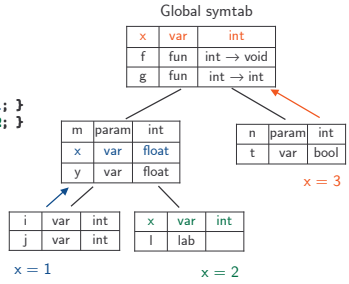
- The hierarchical structure of symbol tables automatically solves the problem of shadowing (identifiers with the same name declared in inner scopes)
- To find which is the declaration of an identifier that is active at a program point :
  - Start from the current scope
  - Go up in the hierarchy until you find an identifier with the same name

Example

```
int x;

void f(int m) {
    float x, y;
    ...
   { int i, j; x = 1; }
   { int x; l: x = 2; }
}

int g(int n) {
    bool t;
    x = 3;
}
```

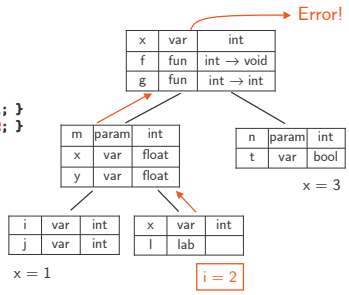


Catching Semantic Errors

```
int x;

void f(int m) {
    float x, y;
    ...
   { int i, j; x = 1; }
   { int x; l: i = 2; }
}

int g(int n) {
    bool t;
    x = 3;
}
```

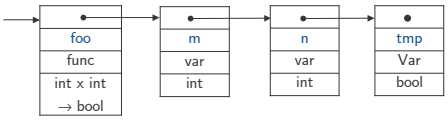


Symbol Table Operations

- Two operations:
  - An insert operation adds new identifiers in the table
  - A lookup function searches symbols by name
- Cannot build symbol tables during lexical analysis
  - hierarchy of scopes encoded in the syntax
- Build the symbol tables:
  - while parsing, using the semantic actions
  - After the AST is constructed

Implementation 1

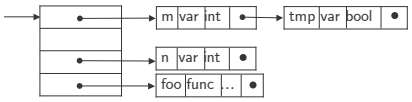
- Linear dynamic structure: `java.util.List, java.util.Vector`
  - One cell per entry in the table
  - Simple structure, grows dynamically



- Disadvantage: inefficient (i.e., slow) for large symbol tables
  - need to scan half the structure on average

Implementation 2

- Efficient lookup implementation: `java.util.HashMap`
  - It is an array of lists (buckets)
  - Uses a hashing function to map the symbol name to the corresponding bucket: hashfunc : string → int
  - Good hash function = even distribution in the buckets



- Disadvantage: structure complexity and space overhead is not justified for small sets of identifiers

Forward References

- Forward references = use an identifier within the scope of its declaration, but before it is declared
- Two-pass approach:
  - Record declarations in the first pass
  - Check uses in the second pass
- Example:

```
class A {
   int m() { return n(); }
   int n() { return 1; }
}
```