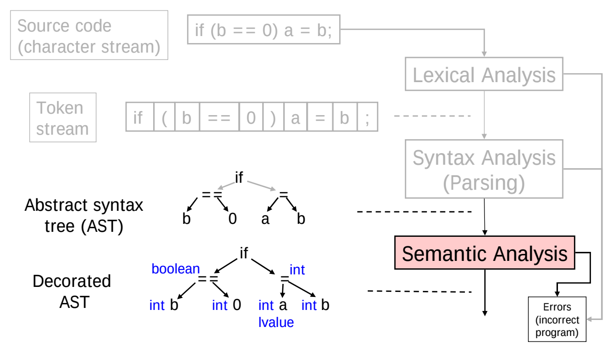
Symbol Tables

Where We Are



Non-Context-Free Syntax

- Programs that are correct with respect to the language's lexical and context-free syntactic rules may still contain other errors
- Lexical analysis and context-free syntax analysis are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc.
- Non-context-free syntactic analysis is known as semantic analysis

Incorrect Programs

- Example 1: lexical analysis does not distinguish between different variable or function identifiers (it returns the same token for all identifiers)

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

- Example 2: syntax analysis does not correlate the declarations with the uses of variables in the program:

int a;	
a = 1;	a = 1;

- Example 3: syntax analysis does not correlate the types from the declarations with the uses of variables:

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

Goals of Semantic Analysis

- Semantic analysis ensures that the program satisfies a set of additional rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - Variables must be declared before being used
  - A variable should not be declared multiple times in the same scope
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The condition of an if-statement must have type Boolean
- Some categories of rules:
  - Semantic rules regarding types
  - Semantic rules regarding scopes

Type Information

- Type information classifies a program's constructs (e.g., variables, statements, expressions, functions) into categories, and imposes rules on their use (in terms of those categories) with the goal of avoiding runtime errors

variables:	int a;	integer location
expressions:	(a+1) == 2	Boolean
statements:	a = 1.0;	void
functions:	int pow(int n, int m)	int x int → int

Type Checking

- Type checking is the validation of the set of type rules
- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has Boolean type

Type Checking

- More examples:
  - For each assignment statement, the type of the updated variable must match the type of the expression being assigned
  - For each call statement foo(v_1, ..., v_n), the type of each actual argument v_i must match the type of the corresponding formal parameter f_i from the declaration of function foo
  - The type of the return value must match the return type from the declaration of the function
- Type checking: next two lectures.

Scope Information

- Scope information characterizes the declaration of identifiers and the portions of the program where use of each identifier is allowed
  - Example identifiers: variables, functions, objects, labels
- Lexical scope is a textual region in the program
  - Statement block
  - Formal argument list
  - Object body
  - Function or method body
  - Module body
  - Whole program (multiple modules)
- Scope of an identifier: the lexical scope in which it is valid

Scope Information

- Scope of variables in statement blocks:

```
{ int a;              } ← scope of variable a
  ...
   { int b;  }  ← scope of variable b
   }
  ...
}
```

- In C:
  - Scope of file static variables: current file
  - Scope of external variables: whole program
  - Scope of automatic variables, formal parameters, and function static variables: the function

Scope Information

- Scope of formal arguments of functions/methods:

```
int factorial(int n) {      } ← scope of formal parameter n
   ...
}
```

- Scope of labels:

```
void f() {
   ... goto l; ...          } ← scope of label l
   l: a =1;
   ... goto l; ...
}
```


Scope Information

- Scope of object fields and methods:

```
class A {
   private int x;                    } ← scope of field x
   public void g() { x=1; }
   ...
}
class B extends A {                  } ← scope of method g
   ...
   public int h() { g(); }
   ...
}
```

Semantic Rules for Scopes

- Main rules regarding scopes:
  Rule 1: Use an identifier only if defined in enclosing scope
  Rule 2: Do not declare identifiers of the same kind with identical names more than once in the same 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

- Semantic checks refer to properties of identifiers in the program -- their scope or type
- Need an environment to store the information about identifiers = symbol table
- Each entry in the symbol table contains
  - the name of an identifier
  - additional information: its kind, its type, if it is constant, ...

NAME	KIND	TYPE	OTHER
foo	fun	int x int → bool	extern
m	par	int	auto
n	par	int	const
tmp	var	bool	const

Scope Information

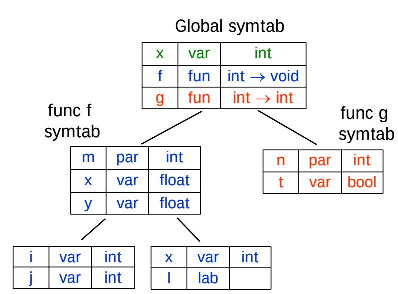
- How to represent scope information in the symbol table?
- Idea:
  - There is a hierarchy of scopes in the program
  - Use a similar hierarchy of symbol tables
  - One symbol table for each scope
  - Each symbol table contains the symbols declared in that lexical scope

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

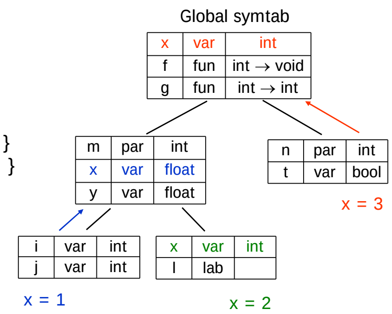
- The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes)
- To find the declaration of an identifier that is active at a program point:
  - Start from the current scope
  - Go up in the hierarchy until you find an identifier with the same name, or fail

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;
}
```

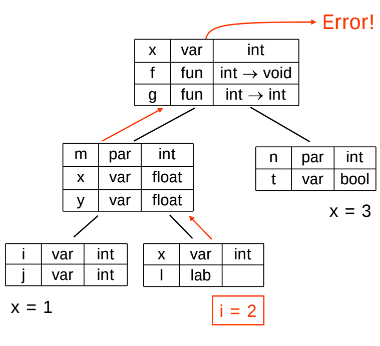


Catching Semantic Errors

```
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

- Three operations
  - Create a new empty symbol table with a given parent table
  - Insert a new identifier in a symbol table (or error)
  - Lookup an identifier in a symbol table (or error)
- 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

Array Implementation

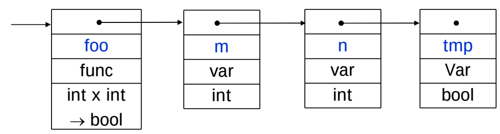
- Simple implementation = array
  - One entry per symbol
  - Scan the array for lookup, compare name at each entry

foo	fun	int x int → bool
m	arg	int
n	arg	int
tmp	var	bool

- Disadvantage:
  - table has fixed size
  - need to know in advance the number of entries

List Implementation

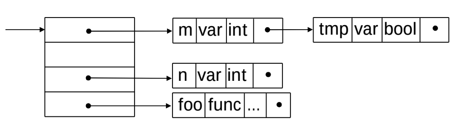
- Dynamic structure = list
  - One cell per entry in the table
  - Can grow dynamically during compilation



- Disadvantage: inefficient for large symbol tables
  - need to scan half the list on average

Hash Table Implementation

- Efficient implementation = hash table
  - 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



- hashfunc("m") = 0, hashfunc("foo") = 3

Forward References

- Forward references = use an identifier within the scope of its declaration, but before it is declared
- Any compiler phase that uses the information from the symbol table must be performed after the table is constructed
- Cannot type-check and build symbol table at the same time
- Example (requiring 2 passes):

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

Summary

- Semantic checks ensure the correct usage of variables, objects, expressions, statements, functions, and labels in the program
- Scope semantic checks ensure that identifiers are correctly used within the scope of their declaration
- Type semantic checks ensures the type consistency of various constructs in the program
- Symbol tables: a data structure for storing information about symbols in the program
  - Used in semantic analysis and subsequent compiler stages
- Next time: type-checking