Semantic Analysis I
Syntax Directed Definition
Symbol Tables

EECS 483 – Lecture 11
University of Michigan
Wednesday, October 11, 2006

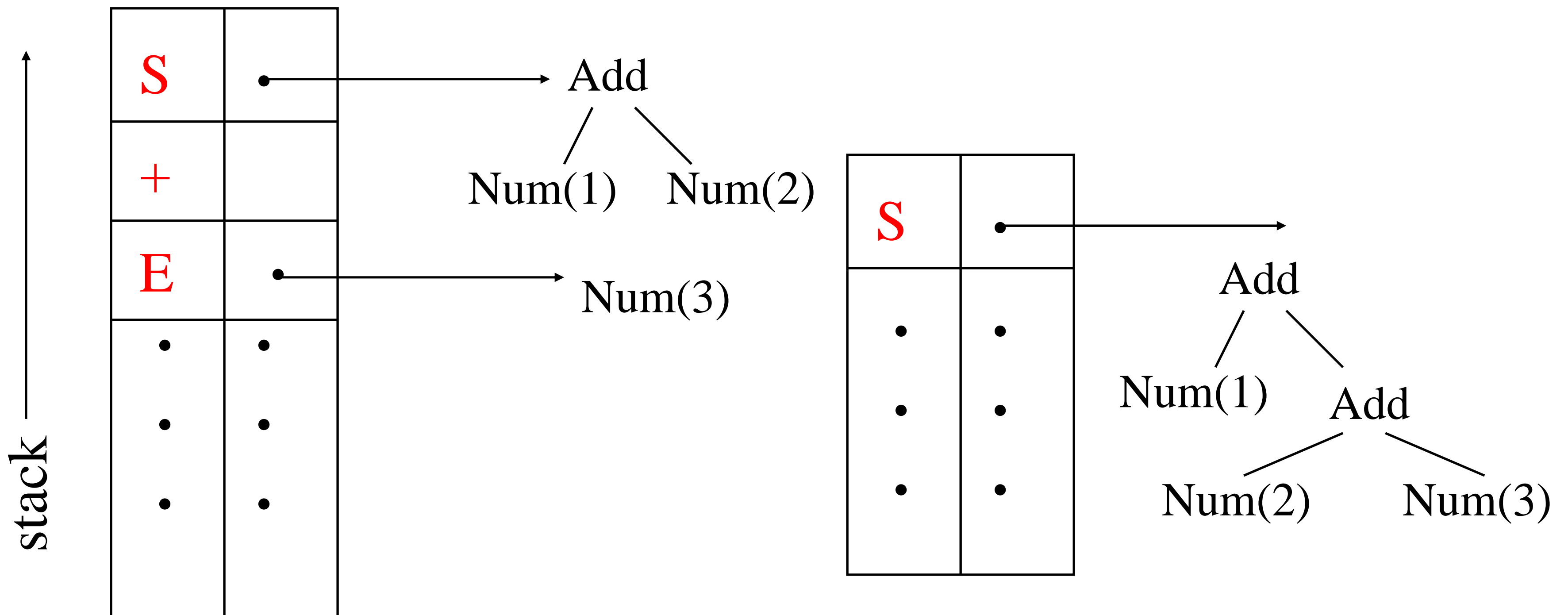
Announcements

- Updated schedule (week behind syllabus)
  - Today 10/11: Semantic analysis I
  - Mon 10/16: Fall break – no class
  - Wed 10/18: Semantic analysis II
  - Mon 10/23: MIRV Q/A session (Yuan Lin)
  - Wed 10/25: Semantic analysis III (Simon Chen)
  - Mon 10/30: Exam review
  - Wed 11/1: Exam 1 in class
- Project 2
  - Teams of 2 → Please send Simon/I mail with names
    - Persons can work individually if really want to
  - No extensions on deadline due to Exam, so get started!
- Reading - 5.1-5.6, 7.6

From Last Time: AST Construction for LR

S → E + S | S
E → num | (S)

input string: “1 + 2 + 3”



Before reduction: S → E + S

After reduction: S → E + S

Problems

- Unstructured code: mixing parsing code with AST construction code
- Automatic parser generators
  - The generated parser needs to contain AST construction code
  - How to construct a customized AST data structure using an automatic parser generator?
- May want to perform other actions concurrently with parsing phase
  - E.g., semantic checks
  - This can reduce the number of compiler passes

Syntax-Directed Definition

- Solution: Syntax-directed definition
  - » Extends each grammar production with an associated semantic action (code):
    - S → E + S {action}
  - » The parser generator adds these actions into the generated parser
  - » Each action is executed when the corresponding production is reduced

Semantic Actions

- Actions = C code (for bison/yacc)
- The actions access the parser stack
  - » Parser generators extend the stack of symbols with entries for user-defined structures (e.g., parse trees)
- The action code should be able to refer to the grammar symbols in the productions
  - » Need to refer to multiple occurrences of the same non-terminal symbol, distinguish RHS vs LHS occurrence
    - $E \rightarrow E + E$
  - » Use dollar variables in yacc/bison ($$, $1, $2, etc.)
    - expr ::= expr PLUS expr {$$ = $1 + $3;}

Building the AST

- Use semantic actions to build the AST
- AST is built bottom-up along with parsing

Recall: User-defined type for objects on the stack (%union)

```
expr ::= NUM                  {$$ = new Num($1.val); }
expr ::= expr PLUS expr       {$$ = new Add($1, $3); }
expr ::= expr MULT expr       {$$ = new Mul($1, $3); }
expr ::= LPAR expr RPAR       {$$ = $2; }
```

Class Problem

E → num | (E) | E + E | E * E

Assume left associative

Perform a LR derivation of the string: "(1+2)*3"
Show where each part of the AST is constructed

Other Syntax-Directed Definitions

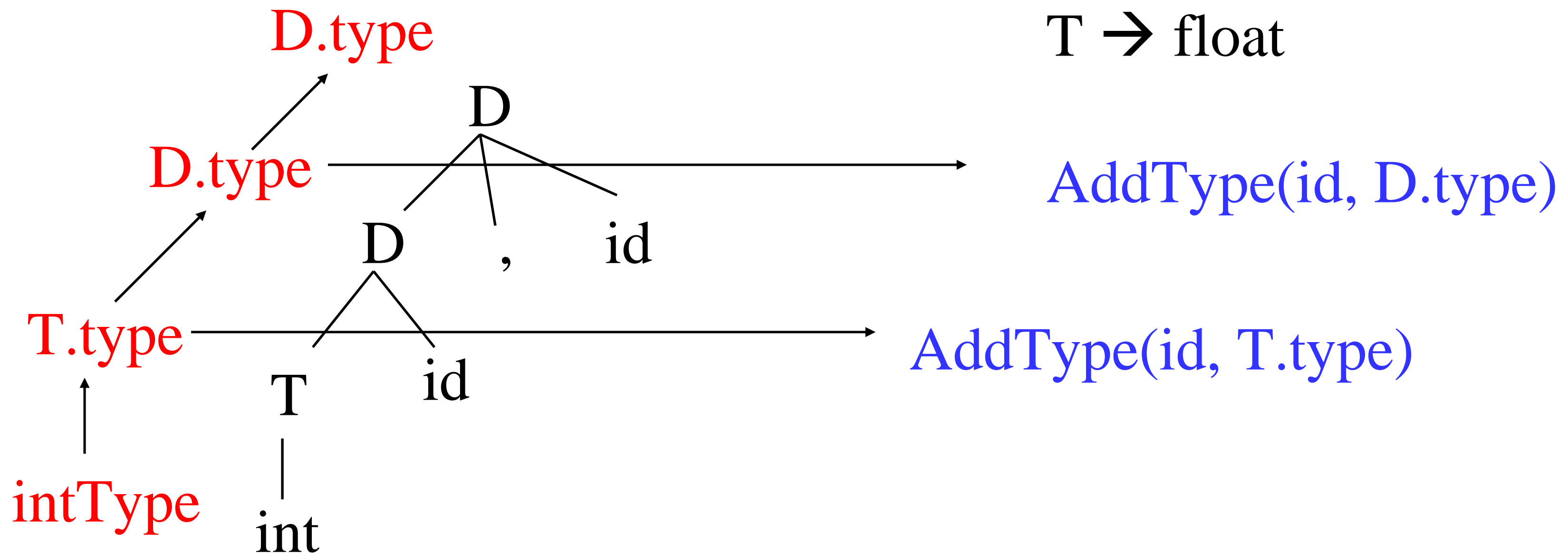
- Can use syntax-directed definitions to perform semantic checks during parsing
  - E.g., type checks
- Benefit = efficiency
  - One single compiler pass for multiple tasks
- Disadvantage = unstructured code
  - Mixes parsing and semantic checking phases
  - Performs checks while AST is changing

Type Declaration Example

Propagate type attributes while building AST from the bottom to the top

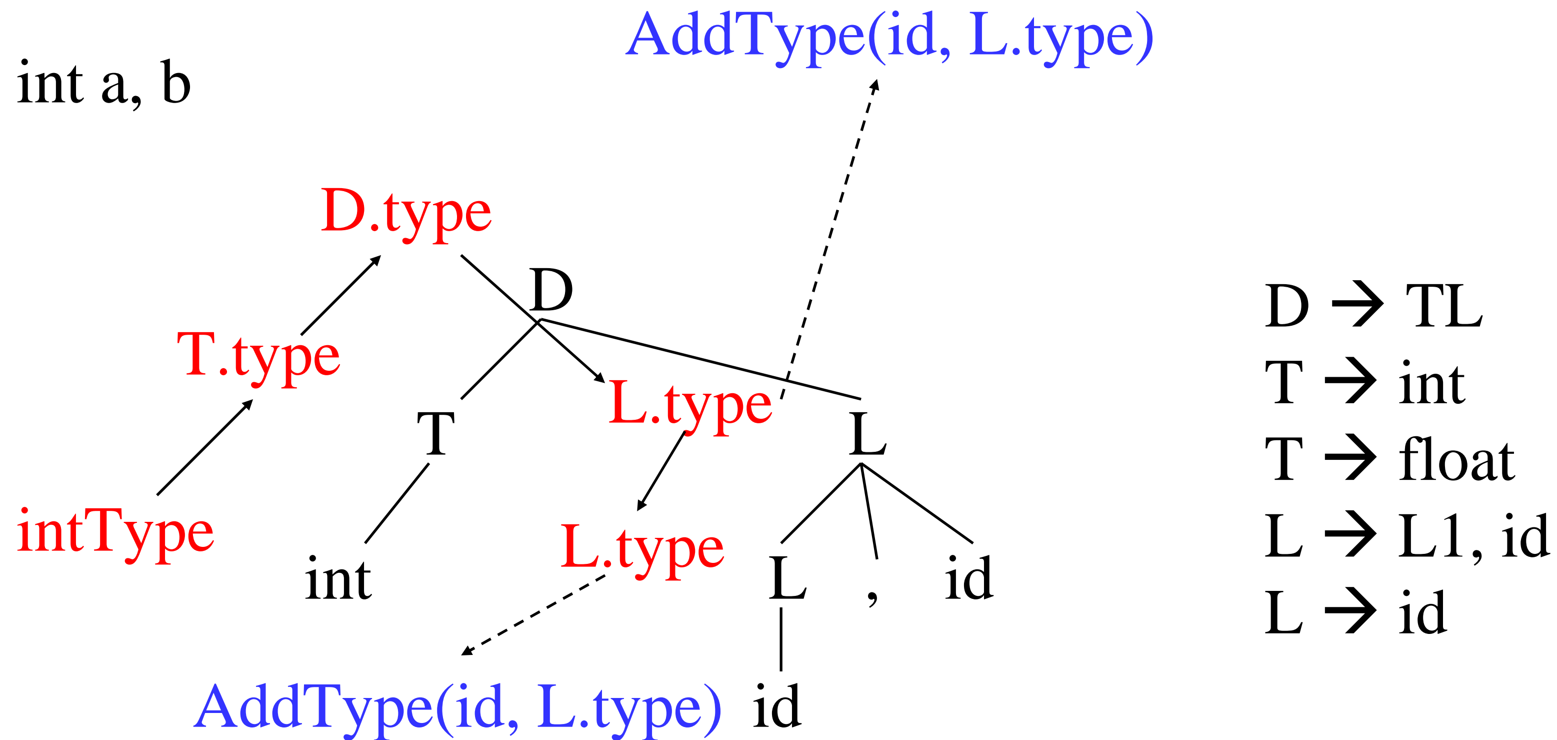
int a, b

D → T id
D → D1, id
T → int
T → float



Type Declaration Example 2

Propagate values both bottom-up and top-down



D → TL
T → int
T → float
L → L1, id
L → id

AST Attributes

- Each node in AST decorated with attributes describing properties of the node
  - » Semantic analysis = compute the attributes of the tree and check the consistency of definitions
- 2 kinds of attributes
  - » Inherited attributes – carry contextual information (variable position info – LHS vs RHS, etc)
  - » Synthesized attributes – modify context (by declaring variables, etc.) and produce code lists (instructions representing operations performed in sub-tree)

AST Attributes (2)

- An attribute for a node in the AST depends on values from parent nodes, sibling nodes and children nodes for evaluation
  - » Values from parents and siblings = inherited
  - » Values from children = synthesized
- Terminals compute only synthesized attrs
- Non-terminals may compute either
  - » May compute inherited attrs from its children and pass these values down the parse tree
  - » May compute synthesized attribute and pass these values up the parse tree
- Constant values called intrinsic attributes

Strategies for Attribute Evaluation

- Walk dependence tree
  - » Construct AST, use that to establish the dependence relationships to guide attribute evaluation
  - » Most flexible, but may fail if get cycle
  - » Build dep graph, topo sort determines order
- Rules based
  - » Order of evaluation of attributes established when the compiler is constructed
- On-the-fly
  - » Order determined by order nodes are visited (e.g., parsing method, top-down or bottom-up)

On-the-fly Evaluation

- Most efficient, but only works with restrictive forms of attributes
  - » L-attributed – RHS symbol depends only upon inherited symbols of LHS and synthesized attributes of symbols to the left of it in the production, and synthesized attributes of the LHS depend only upon inherited attributes of LHS and attributes of RHS
    - Attribute info flows from Left to right
    - Depth-first traversal will suffice
  - » S-attributed – Only synthesized attributes, node's attributes only dependent on attributes on stack
    - Evaluate bottom-up

Multi-Pass Approach

- Separate AST construction from semantic checking phase
- Traverse the AST and perform semantic checks (or other actions) only after the tree has been built and its structure is stable
- This approach is less error-prone
  - It is better when efficiency is not a critical issue
- Attribute evaluation proceeds as tree-walk of the AST

Semantic Analysis

- Lexically and syntactically correct programs may still contain other errors
- Lexical and syntax analyses are not powerful enough to ensure the correct usage of variables, objects, functions, ...
- Semantic analysis: Ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements)

Class Problem

Classify each error as lexical, syntax, semantic, or correct.

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

```
1int x;
x = 2;
```

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

```
{
   int a;
   a = 1;
}
{
   a = 2;
}
```

```
in a;
a = 1;
```

```
int foo(int a)
{
 foo = 3;
}
```

```
int foo(int a)
{
   a = 3;
}
```

Categories of Semantic Analysis

- Examples of semantic rules
  - Variables must be defined before being used
  - A variable should not be defined multiple times
  - In an assignment stmt, the variable and the expression must have the same type
  - The test expr. of an if statement must have boolean type
- 2 major categories
  - Semantic rules regarding types
  - Semantic rules regarding scopes

Type Information/Checking

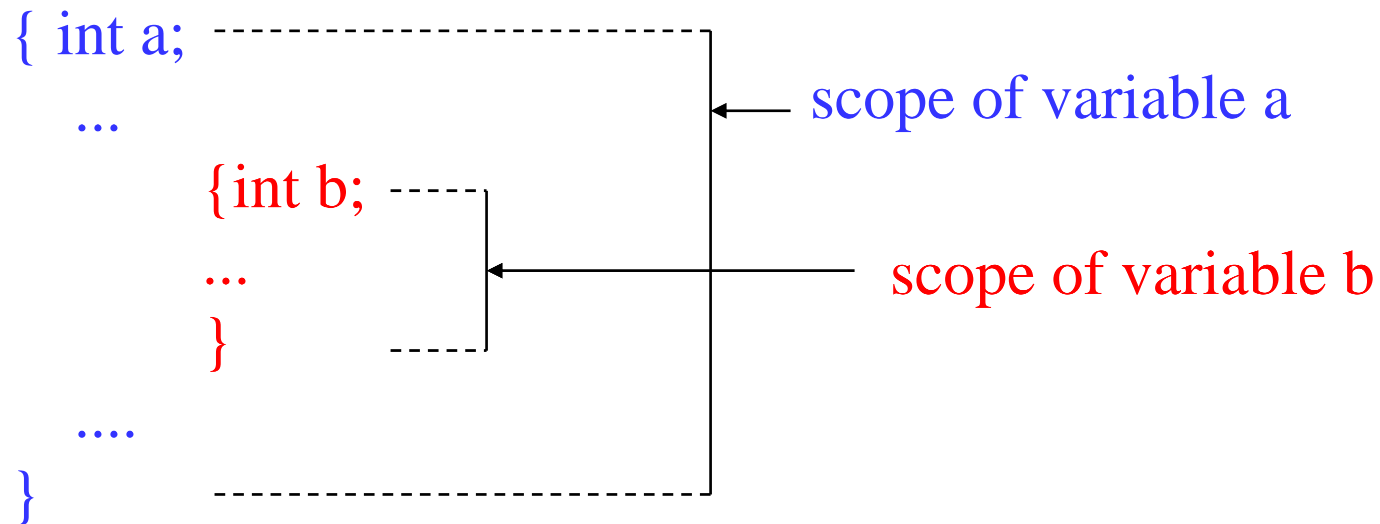
- Two main categories of semantic analysis
  - Type information
  - Scope information
- Type Information: Describes what kind of values correspond to different constructs: variables, statements, expressions, functions, etc.
  - variables: int a; integer
  - expressions: (a+1) == 2 boolean
  - statements: a = 1.0; floating-point
  - functions: int pow(int n, int m) int = int,int
- Type Checking: Set of rules which ensures the type consistency of different constructs in the program

Scope Information

- Characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier
  - Example identifiers: variables, functions, objects, labels
- Lexical scope: textual region in the program
  - Examples: Statement block, formal argument list, object body, function or method body, source file, whole program
- Scope of an identifier: The lexical scope its declaration refers to

Variable Scope

- Scope of variables in statement blocks:



- Scope of global variables: current file
- Scope of external variables: whole program

Function Parameter and Label Scope

- Scope of formal arguments of functions:

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

scope of argument n

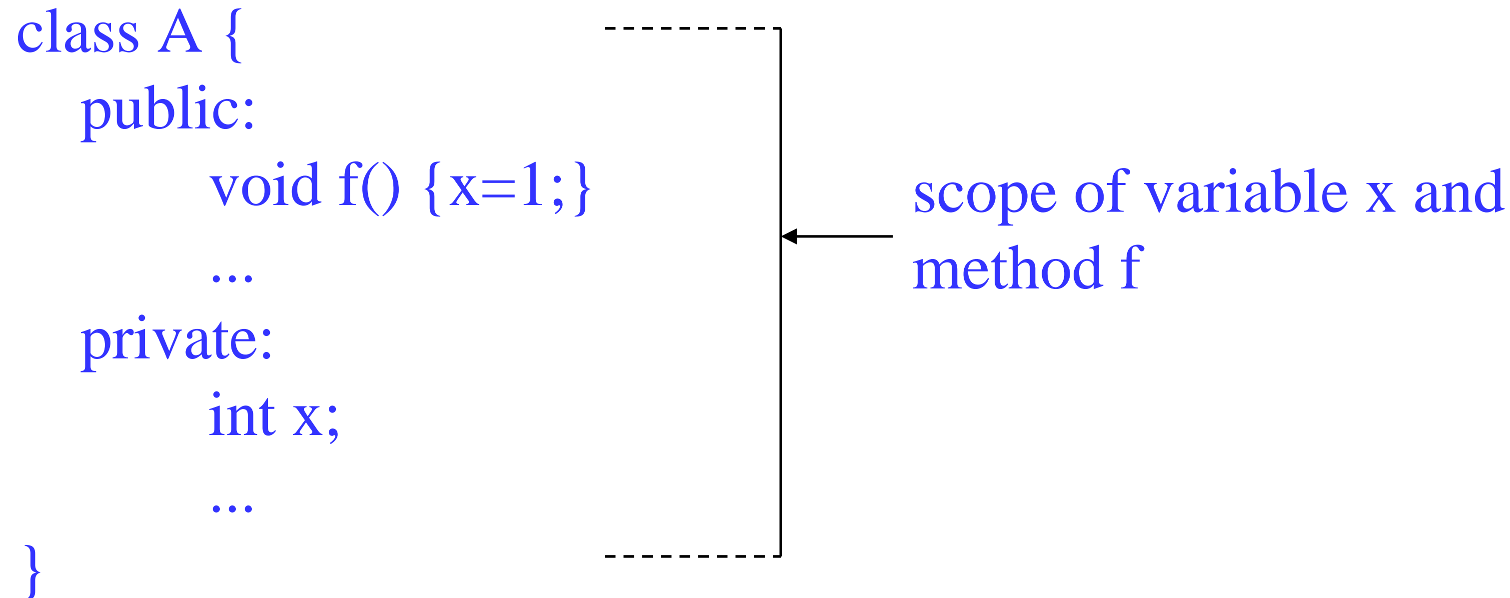
- Scope of labels:

```
void foo() {
    ... goto lab;
    ...
    lab: i++;
    ... goto lab;
    ...
}
```

scope of label lab, Note in Ansi-C all labels have function scope regardless of where they are

Scope in Class Declaration

- Scope of object fields and methods:



Semantic Rules for Scopes

- Main rules regarding scopes:
  - Rule 1: Use each identifier only within its scope
  - Rule 2: Do not declare identifier of the same kind with identical names more than once in the same lexical scope

```
class X {
   int X;
   void X(int X) {
         X: ...
         goto X;
   }
}
```

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

Are these legal? If not, identify the illegal portion.

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:
  - Name of an identifier
  - Additional info about identifier: kind, type, constant?

NAME	KIND	TYPE	ATTRIBUTES
foo	func	int,int → int	extern
m	arg	int	
n	arg	int	const
tmp	var	char	const

Scope Information

- How to capture the scope information in the symbol table?
- Idea:
  - » There is a hierarchy of scopes in the program
  - » Use 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) {
    char t;
    ... ;
}
```

Global symtab

x	var	int
f	func	int → void
g	func	int → int

func f symtab

m	arg	int
x	var	float
y	var	float

func g symtab

n	arg	int
t	var	char

i	var	int
j	var	int

x	var	int
l	label	

Identifiers with Same Name

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

Class Problem

Associate each definition of x with its appropriate symbol table entry

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

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

Global symtab

x	var	int
f	func	int → void
g	func	int → int

Child of global (f):

m	arg	int
x	var	float
y	var	float

Child of global (g):

n	arg	int
t	var	char

Children of f's table:

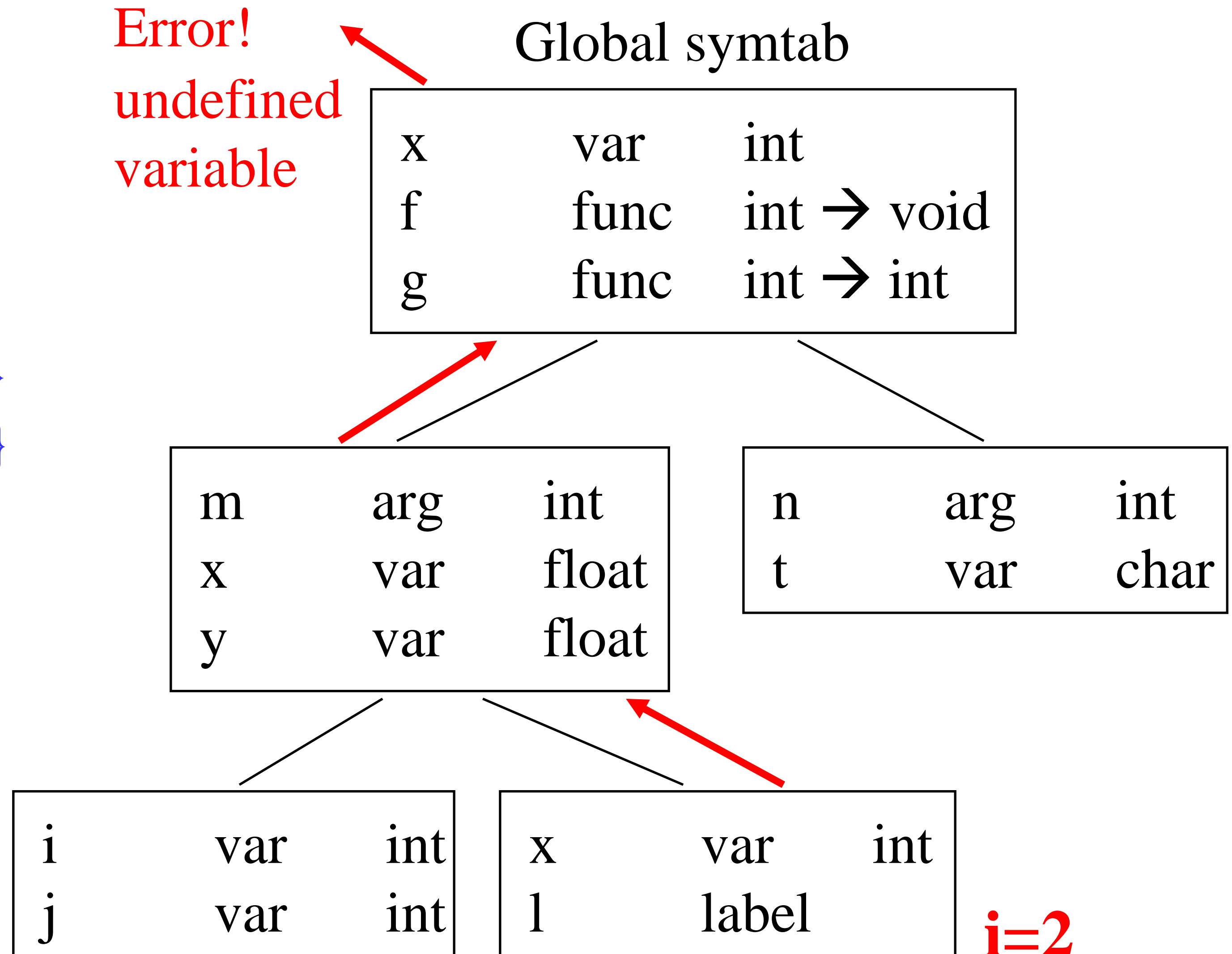
i	var	int
j	var	int

x	var	int
l	label	

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) {
    char t;
    x=3;
}
```



Symbol Table Operations

- Two operations:
  - » To build symbol tables, we need to insert new identifiers in the table
  - » In the subsequent stages of the compiler we need to access the information from the table: use lookup function
- Cannot build symbol tables during lexical analysis
  - » Hierarchy of scopes encoded in syntax
- Build the symbol tables:
  - » While parsing, using the semantic actions
  - » After the AST is constructed

Forward References

- Use of 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

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