Semantic Analysis II
Type Checking

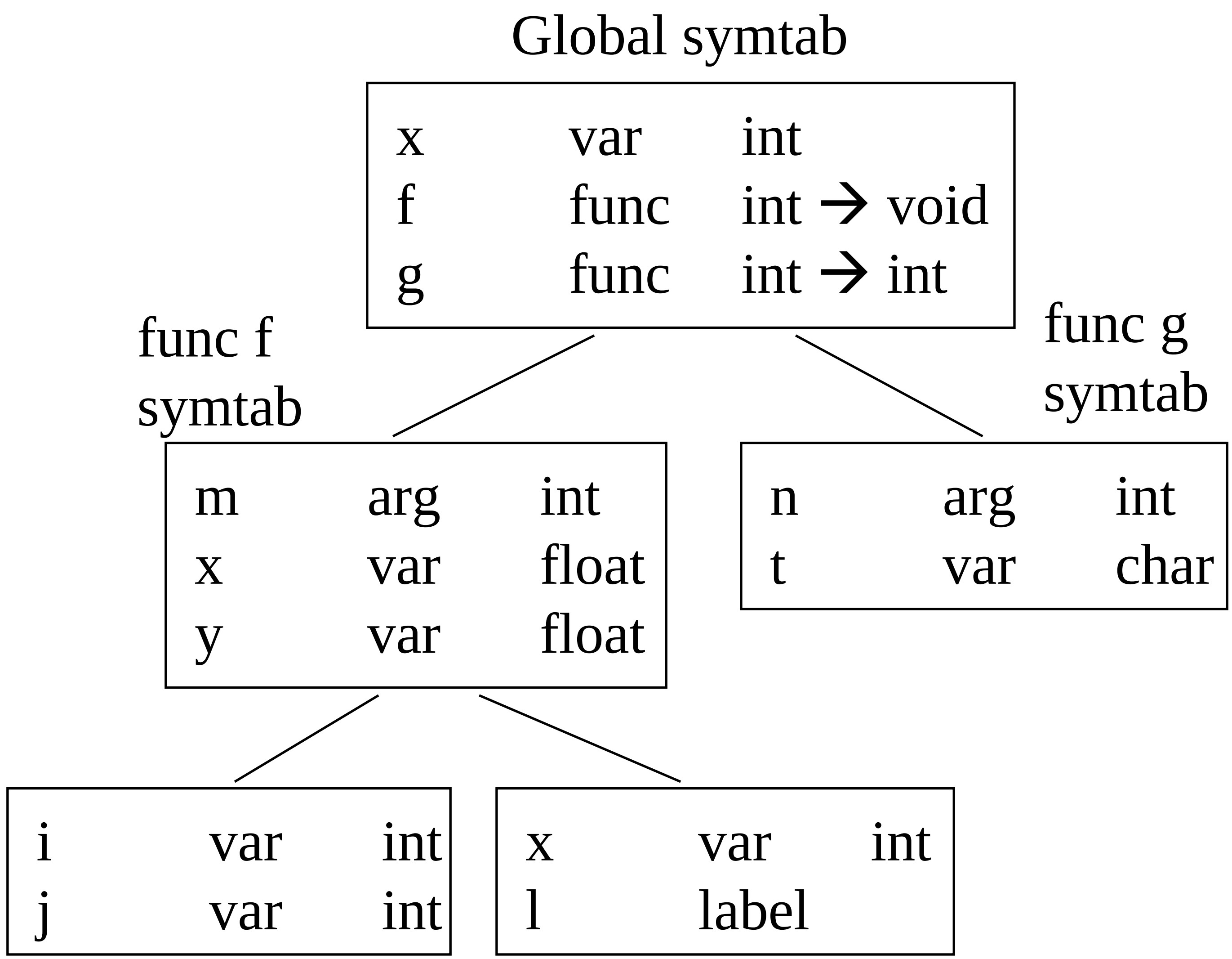
EECS 483 – Lecture 12
University of Michigan
Wednesday, October 18, 2006

Announcements

- Updated schedule
  - Today: 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 - 6.1-6.4

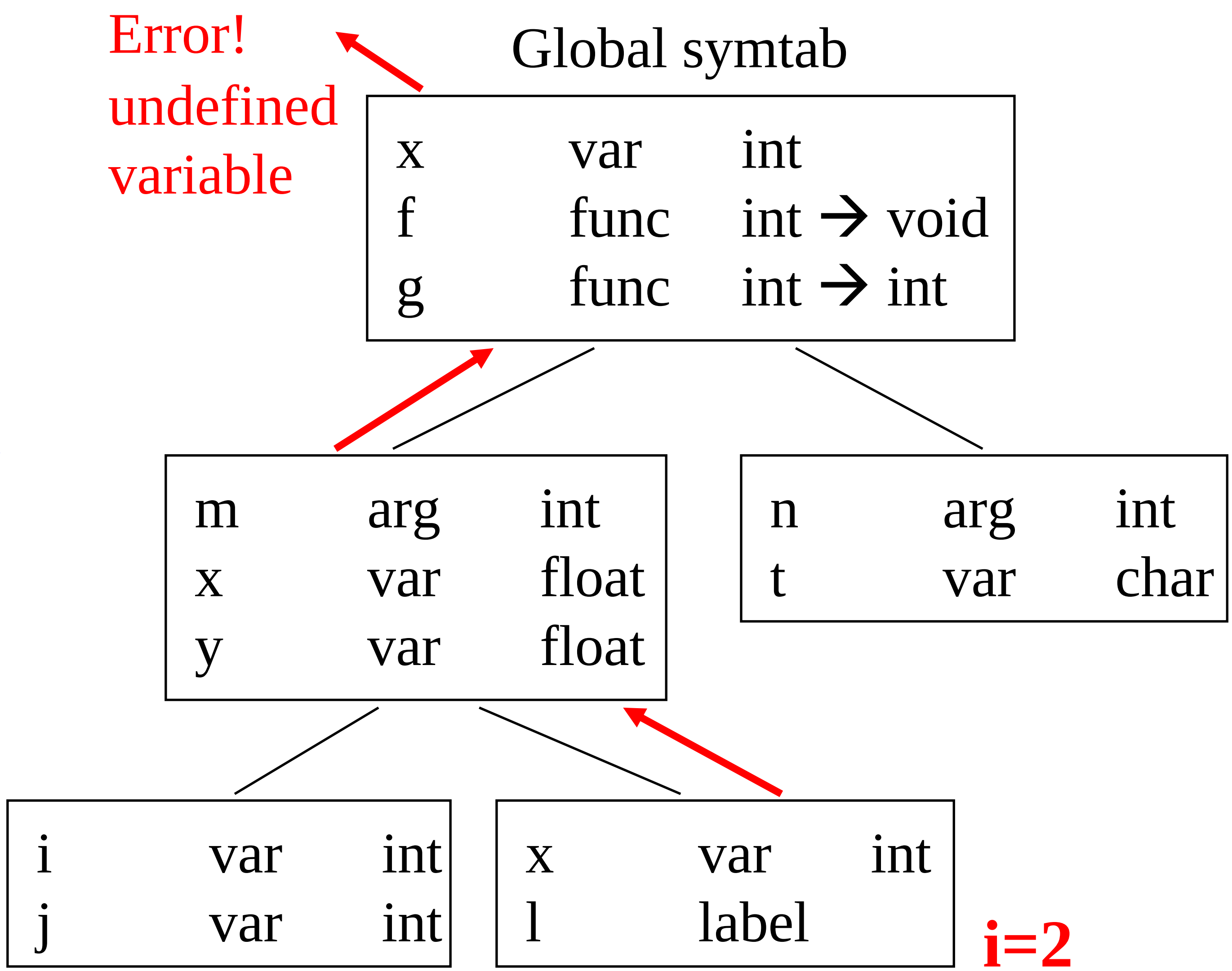
From Last Time – Hierarchical Symbol Table

```
int x;

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

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



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; }
}
```

Back to Type Checking

- What are types?
  - » They describe the values computed during the execution of the program
  - » Essentially they are a predicate on values
    - E.g., "int x" in C means $-2^{31} <= x < 2^{31}$
- Type Errors: improper or inconsistent operations during program execution
- Type-safety: absence of type errors

How to Ensure Type-Safety

- Bind (assign) types, then check types
- Type binding: defines type of constructs in the program (e.g., variables, functions)
  - » Can be either explicit (int x) or implicit (x=1)
  - » Type consistency (safety) = correctness with respect to the type bindings
- Type checking: determine if the program correctly uses the type bindings
  - » Consists of a set of type-checking rules

Type Checking

- Semantic checks to enforce the type safety of the program
- Examples
  - » Unary and binary operators (e.g. +, ==, [ ]) must receive operands of the proper type
  - » Functions must be invoked with the right number and type of arguments
  - » Return statements must agree with the return type
  - » In assignments, assigned value must be compatible with type of variable on LHS
  - » Class members accessed appropriately

4 Concepts Related to Types/Languages

1. Static vs dynamic checking
   » When to check types
2. Static vs dynamic typing
   » When to define types
3. Strong vs weak typing
   » How many type errors
4. Sound type systems
   » Statically catch all type errors

Static vs Dynamic Checking

- Static type checking
  - Perform at compile time
- Dynamic type checking
  - Perform at run time (as the program executes)
- Examples of dynamic checking
  - Array bounds checking
  - Null pointer dereferences

Static vs Dynamic Typing

- Static and dynamic typing refer to type definitions (i.e., bindings of types to variables, expressions, etc.)
- Static typed language
  - » Types defined at compile-time and do not change during the execution of the program
    - C, C++, Java, Pascal
- Dynamically typed language
  - » Types defined at run-time, as program executes
    - Lisp, Smalltalk

Strong vs Weak Typing

- Refer to how much type consistency is enforced
- Strongly typed languages
  - Guarantee accepted programs are type-safe
- Weakly typed languages
  - Allow programs which contain type errors
- These concepts refer to run-time
  - Can achieve strong typing using either static or dynamic typing

Soundness

- Sound type systems: can statically ensure that the program is type-safe
- Soundness implies strong typing
- Static type safety requires a conservative approximation of the values that may occur during all possible executions
  - » May reject type-safe programs
  - » Need to be expressive: reject as few type-safe programs as possible

Class Problem

Classify the following languages: C, C++, Pascal, Java, Scheme ML, Postscript, Modula-3, Smalltalk, assembly code

	Strong Typing	Weak Typing
Static Typing		
Dynamic Typing		

Why Static Checking?

- Efficient code
  - » Dynamic checks slow down the program
- Guarantees that all executions will be safe
  - » Dynamic checking gives safety guarantees only for some execution of the program
- But is conservative for sound systems
  - » Needs to be expressive: reject few type-safe programs

Type Systems

- What are types?
  - They describe the values computed during the execution of the program
  - Essentially they are a predicate on values
    - E.g., “int x” in C means –2^31 <= x < 2^31
- Type expressions: Describe the possible types in the program
  - E.g., int, char*, array[], object, etc.
- Type system: Defines types for language constructs
  - E.g., expressions, statements

Type Expressions

- Language type systems have basic types (aka: primitive types or ground types)
  - E.g., int, char*, double
- Build type expressions using basic types:
  - Type constructors
    - Array types
    - Structure/object types
    - Pointer types
  - Type aliases
  - Function types

Type Comparison

- Option 1: Implement a method T1.Equals(T2)
  - » Must compare type trees of T1 and T2
  - » For object-oriented languages: also need sub-typing, T1.SubtypeOf(T2)
- Option 2: Use unique objects for each distinct type
  - » Each type expression (e.g., array[int]) resolved to same type object everywhere
  - » Faster type comparison: can use ==
  - » Object-oriented: check subtyping of type objects

Creating Type Objects

- Build types while parsing – use a syntax-directed definition

```
non terminal Type type
type : INTEGER
          {$$ = new IntType(id); }
      | ARRAY LBRACKET type RBRACKET
          {$$ = new ArrayType($3); } ;
```

- Type objects = AST nodes for type expressions

Processing Type Declarations

- Type declarations add new identifiers and their types in the symbol tables
- Class definitions must be added to symbol table:
  - » class_defn : CLASS ID {decls} ;
- Forward references require multiple passes over AST to collect legal names
  - » class A {B b; }
  - » class B { ... }

Type Checking

- Type checking = verify typing rules
  - » E.g., “Operands of + must be integer expressions; the result is an integer expression”
- Option 1: Implement using syntax-directed definitions (type-check during the parsing)

```
expr:   expr PLUS expr {
        if ($1 == IntType && $3 == IntType)
                $$ = IntType
        else
                TypeCheckError(“+”);
        }
```

Type Checking (2)

- Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:

```
class Add extends Expr {
        Type typeCheck() {
                Type t1 = e1.typeCheck(), t2 = e2.typeCheck();
                if (t1 == Int && t2 == Int) return Int
                else TypeCheckError(“+”);
        }
}
```

Type Checking Identifiers

- Identifier expressions: Lookup the type in the symbol table

```
class IdExpr extends Expr {
        Identifier id;
        Type typeCheck() {return id.lookupType(); }
}
```

Next Time: Static Semantics

- Can describe the types used in a program
- How to describe type checking
- Static semantics: Formal description for the programming language
- Is to type checking:
  - As grammar is to syntax analysis
  - As regular expression is to lexical analysis
- Static semantics defines types for legal ASTs in the language