CS42/413

Introduction to Compilers
Radu Rugina

Lecture 11: Symbol Tables
12 Feb 03

Abstract Syntax Trees
- 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

Visitors
- Visitor pattern: very useful object-oriented programming pattern that separates code of a data structure from code which traverses the structure
- Use visitors for walking the AST
  - Traversal code not embedded in the AST node classes
  - Implement each traversal as a separate class hierarchy
- Define a Visitor interface for all visitor classes
- Extend each class in the structure with a method that accepts any visitor

A Visitor Methodology

```java
class Expr {
  public void accept(Visitor v) {
    v.visit(this);
  }
}
class binaryExpr extends Expr {
  public void accept(Visitor v) {
    left.accept(v); right.accept(v);
    v.visit(this);
  }
}
class unaryExpr extends Expr {
  public void accept(Visitor v) {
    child.accept(v); v.visit(this);
  }
}
```

Visiting the Structure
- For each particular kind of traversal, implement the Visitor interface
  ```java
class TypeCheckVisitor implements Visitor {
  void visit(Expr e) { /* code */ }
  void visit(binaryExpr e) { /* code */ }
  void visit(unaryExpr e) { /* code */ }
}
```
- To traverse expression e:
  ```java
  TypeCheckVisitor v = new TypeCheckVisitor();
  e.accept(v);
  ```

Where We Are

- Source code (character stream)
- Lexical Analysis
- Syntax Analysis (Parsing)
- Semantic Analysis
Incorrect Programs

- Lexically and syntactically correct programs may still contain other 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 or function identifiers (it returns the same token for all identifiers)
  
  \[
  \begin{align*}
  \text{int } a; & \quad \text{int } a; \\
  a = 1; & \quad b = 1;
  \end{align*}
  \]

Incorrect Programs

- Example 2: syntax analysis does not correlate the declarations with the uses of variables in the program:
  
  \[
  \begin{align*}
  \text{int } a; & \quad \text{int } a; \\
  a = 1; & \quad a = 1;
  \end{align*}
  \]

- Example 3: syntax analysis does not correlate the types from the declarations with the uses of variables:
  
  \[
  \begin{align*}
  \text{int } a; & \quad \text{int } a; \\
  a = 1; & \quad a = 1.0;
  \end{align*}
  \]

Goals of Semantic Analysis

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

Type Information

- Type information = describes what kind of values correspond to different constructs: variables, statements, expressions, functions
  
  \[
  \begin{align*}
  \text{variables: } & \quad \text{int } a; \quad \text{integer} \\
  \text{expressions: } & \quad (a+1) == 2 \quad \text{boolean} \\
  \text{statements: } & \quad a = 1.0 \quad \text{floating-point} \\
  \text{functions: } & \quad \text{int pow(int } n, \text{ int } m) \quad \text{int } x \times \text{int } \rightarrow \text{int}
  \end{align*}
  \]

Type Checking

- Type checking = set of rules which ensures the type consistency of different constructs in the program
- 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 \text{foo} (v_1, ..., v_n), the type of each actual argument \(v_i\) must match the type of the corresponding formal argument \(f\) from the declaration of function \text{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 it is allowed to use each identifier
  - Example identifiers: variables, functions, objects, labels
- Lexical scope = 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 its declaration refers to

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 lexical scope
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds

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, ...
**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

**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 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

**Caught Semantic Errors**

```c
int x;
void f(int m) {
    float x, y;
    ...  
    int i, j;  
    { int x; i = 1; }  
    { int x; i = 2; }
}
int g(int n) {
    bool 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 a lookup function
- **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

<table>
<thead>
<tr>
<th>foo</th>
<th>fun</th>
<th>int x int -&gt; bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>arg</td>
<td>int</td>
</tr>
<tr>
<td>n</td>
<td>arg</td>
<td>int</td>
</tr>
<tr>
<td>tmp</td>
<td>var</td>
<td>bool</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>foo</th>
<th>m</th>
<th>n</th>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>func</td>
<td>var</td>
<td>var</td>
<td>Var</td>
</tr>
<tr>
<td>int x int</td>
<td>var</td>
<td>int</td>
<td>bool</td>
</tr>
</tbody>
</table>

- 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

```
  m var int *     tmp var bool *
  n var int *     foo var bool *
```

- 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:

```java
  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