CS412/CS413
Introduction to Compilers
Tim Teitelbaum
Lecture 12: Symbol Tables
February 20, 2007

Non-Context-Free Syntax
- Programs that are correct with respect to the language’s lexical and context-free syntactic rules may still contain other syntactic 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
- Examples of types:
  - 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(v1, …, vn), the type of each actual argument vi must match the type of the corresponding formal argument fi 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: variable s, 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:
  ```plaintext
  { int a;
  ...
  { int 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:
  ```plaintext
  int factorial(int n) {
  ...
  }
  ```
- Scope of labels:
  ```plaintext
  void f() { 
  ... goto l; ...
  l: a =1; 
  ... goto l; ...
  }
  ```
- Scope of object fields and methods:
  ```plaintext
  class A {
  private int x;
  public void g() { x=1; }
  ...
  }
  class B extends A {
  ...
  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 lexical scope
  
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds

```java
class X {
    int X;
    void X(int X) {
        goto X;
        { int X; break X; }
    }
}
```

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, ...

<table>
<thead>
<tr>
<th>NAME</th>
<th>KIND</th>
<th>TYPE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>fun</td>
<td>int</td>
<td>int → bool</td>
</tr>
<tr>
<td>m</td>
<td>arg</td>
<td>int</td>
<td>auto</td>
</tr>
<tr>
<td>n</td>
<td>arg</td>
<td>int</td>
<td>const</td>
</tr>
<tr>
<td>tmp</td>
<td>var</td>
<td>bool</td>
<td>const</td>
</tr>
</tbody>
</table>

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

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

Example

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
Catching Semantic Errors

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

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

- **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

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

- **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

```c
m var int
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:
  ```c
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** ensure 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.